TDD - pequenos passos fazem toda diferença

Material utilizado na oficina da FLISOL realizada no dia 26/04/2014 na edição de Curitiba - PR, ministrada por Andre Cardoso.

Sobre mim

Andre Luiz Kwasniewski Cardoso

Desenvolvedor Web utilizando linguagem PHP e tecnologias que a complementam como: Html, Css, Sass, Javascript, jQuery, sistema de versionamento Git entre outras.

Apaixonado por software livre, membro ativo de comunidades de desenvolvimento, membro da equipe que está tentando reativar o grupo de desenvolvimento em PHP no estado do Paraná, autor do Tableless, mantenedor do blog andrebian.com, criador de soluções, resolvedor de problemas, louco... enfim um desenvolvedor comum.

Meus contatos:

Email: andrecardosodev@gmail.com
Twitter: @andrebian
Google +: http://goo.gl/4dzNUA
Facebook: /andrebiancardoso
Blog: http://andrebian.com
Github: /andrebian
Tableless: http://tableless.com.br/author/andrecardosodev/

Sumário

Capítulo 1

- Vantagens
- Quais são os passos do TDD?
- Baby Steps
- FIRST

Capítulo 2

- Baixando e instalando
- Primeiro teste

Capítulo 3

- Baixando o ORM Doctrine
- Criando o boostrap da aplicação
- Criando o Bootstrap para testes
- Xml de configurações
- Criando o TestCase
- Criando e testando entidades
- Criando e testando serviços

Capítulo 4

- Conceito
- Instalando
- Casos de uso
- Utilização básica
- Realizando testes com Mocks
- Conclusão

Capítulo 5

- Testes em grupos
- Cobertura
- Diferenças entre tipos de testes
- Diferentes tipos de testes
- Praticando TDD

Referências

Capítulo 1

Com vocês Test Driven Development ou simplesmente TDD.

Introdução

Na oficina “TDD - Pequenos passos fazem toda diferença” foi abordado o tema TDD, sigla para *Test Driven Development* ou em bom português, Desenvolvimento Orientado a Testes. Ao longo deste documento você conhecerá mais sobre esta prática que é indispensável para uma evolução responsável de qualquer software.

Apresentação

TDD é o desenvolvimento de software orientado a testes, ou em inglês, Test Driven Development. Mas mais do que simplesmente testar seu código, TDD é uma cultura, uma forma de arte além é claro de um requisito importantíssimo de segurança ao desenvolver.

Segurança ao desenvolver != Segurança da informação.

Com este “manual” você conhecerá um pouco sobre sua motivação e também saberá os fatores que contribuem e dificultam sua prática.

Contratempos (ou desvantagens ...)

Por mais que pareça estranho, começaremos pelas desvantagens. Isto porque somente elas são levadas em consideração por muitos desenvolvedores e principalmente por gerentes de projetos.

Dificuldade em começar

Apesar de uma extensa e clara documentação, iniciar o desenvolvimento orientado a testes pode ser um trabalho árduo para muitos desenvolvedores pelo simples fato de que geralmente muitos iniciantes tentam praticá-lo em código já existente. Este definitivamente não é o caminho. A principal característica do desenvolvimento orientado a testes é que ele seja ORIENTADO a testes. Em outras palavras o código que realizará sua lógica deve ser criado somente após a criação do teste e isso torna-se algo de difícil aceitação pois ainda não se tem nada e já se faz necessário testar sem muitas vezes sequer saber o que será retornado com a execução de um método.

Curva de aprendizado

Complementando o item anterior, este é outro motivo que faz programadores desistirem do desenvolvimento orientado a testes. Como qualquer nova tecnologia, para a pratica de TDD leva-se um bom tempo dependendo disponibilidade e principalmente da vontade do programador.

Tempo

Engana-se quem pensa que produzirá mais código pelo simples fato de utilizar TDD. O TDD na verdade chega a desacelerar a produção de código. Quando falo em produção de código, me refiro à quantidade de linhas escritas. Mas nisso tudo há vantagens não precisa (e não deve) julgar somente pelas dificuldades e/ou desvantagens.

Cultura e entrosamento

Muito fala-se de TDD no Brasil, mas ao questionarmos programadores de diversas empresas muitos apresentam os motivos citados acima para não utilizá-lo.
Existem sim muitas empresas e programadores que levam a prática a sério e a evangelizam justamente por conhecerem as vantagens que o TDD nos traz. Outro ponto importantíssimo é o entrosamento de sua equipe, esta deve estar ciente dos benefícios que o TDD traz. Mesmo que você trabalhe sozinho, ainda sim há alguma cobrança que pede para que você deixe de desenvolver com testes pelo simples fato de que “era pra ontem”.

Vantagens

Vamos falar de coisa boa, não, não é da Tekpix. Agora você conhecerá quais as principais vantagens de se desenvolver com orientação a testes.

Qualidade do código

Um dos principais ensinamentos, senão o principal, do TDD é que se algo não é possível de ser testado então foi desenvolvido de forma errada. Parece um pouco drástico mas não é. Em pouco tempo utilizando testes o programador percebe mudanças relevantes em sua forma de programar. Em suma o uso de TDD ajuda o programador a elaborar um código com cada vez mais qualidade criando objetos concisos.

Raciocínio

Para que o código torne-se mais conciso, tenha menos acoplamentos e dependências o programador deve forçar seu raciocínio a níveis elevados. É muito difícil criar algo que realmente tenha um bom design. Utilizando TDD o programador praticamente obriga-se a olhar seu código de outra forma normalmente jamais vista antes. Aí é que está a parte legal da coisa toda.

Segurança

Ponto importantíssimo para qualquer software nos dias de hoje. Mas não se engane, não estou falando de segurança da informação e sim de segurança ao desenvolver. Pense em uma situação em que o programador tenha um código que desenvolvera ha cerca de um ano. Como normalmente vivemos em um mundo com inúmeros softwares desenvolvidos ao longo de cada ano, torna-se muito difícil lembrar de tudo a respeito de um que merece nossa atenção em determinado momento. Normalmente deve-se realizar um trabalho bastante cauteloso para nova implementação em um software que encontra-se em produção. Toda e qualquer alteração deve ser minunciosamente testada e garantida que não afetará demais módulos do software. Fazer isto manualmente é realmente complicado pois até então não sabe-se (ou lembra-se) ao certo quem afeta quem no sistema. Com a prática de TDD cada pequeno passo do software está devidamente testado. Ou seja, com este cenário o programador pode realizar qualquer alteração sem medo e sem culpa.

Como cada pequeno passo tomado pelo sistema está testado, ao qualquer módulo ou funcionalidade sofrer alteração com poucos segundos descobre-se se houveram quebras e o melhor de tudo, onde foram essas quebras. Com isso em mãos a correção das quebras torna-se uma tarefa simples sem frustrar o cliente e o usuário.

Trabalho em equipe

Por prover mais segurança o trabalho em equipe torna-se muito mais proveitoso eliminando discussões e dúvidas desnecessárias. Ao entrar no desenvolvimento do projeto o novo desenvolvedor tem apenas o trabalho de entender qual task deve ser realizada e ler os testes das features já desenvolvidas. Ao rodar os testes pela primeira vez o programador descobre se está no caminho de ter um entregável mais rapidamente e com segurança. Existem empresas em que um novo programador tem entregáveis logo no primeiro dia de trabalho. Sem testes normalmente haveria um período de adaptação para prévio entendimento do que há no sistema no momento de seu ingresso ao time de desenvolvimento.

Documentação

Ao criar testes descritivos estes servem como uma excelente documentação para o software. Quando qualquer programador for rodar os testes, basta habilitar o modo verbose que uma “história” é contada eliminando o árduo trabalho de documentar um software onde nos meios tradicionais tende a defasar-se. O problema é que a documentação tradicional raramente segue o mesmo ritmo do desenvolvimento. Com os testes unitários a “documentação” é gerada antes mesmo da nova feature ser implementada e permanece fiel a qualquer alteração.

Testes manuais custam caro!

Pense no seguinte problema: Existe um módulo em nosso sistema que é responsável pelo processamento de pagamento via boleto. Um dia qualquer o cliente solicita que seja adicionada a possibilidade de pagamento com cartão de crédito, passado mais um certo tempo, o cliente solicita que seja adicionada a possibilidade de multiplo pagamento. Ou seja, realizar pagamento com mais de um cartão de crédito.

Pense que para cada uma das novas *features* do sistema existe uma sequência de desenvolvimento, testes, testes, testes, testes, testes… testes. Ou seja, pense no tempo que um desenvolvedor ou uma equipe de testers levariam pra realizar compras testando todos os meios de pagamento (gateway/Banco/API, etc) e em cada meio suas diversas formas afim de garantir que além da nova modalidade de pagamento TUDO que já encontrava-se funcionando anteriormente no sistema continue funcionando após tal implementação. Selecionar produto, quantidade, variações, adicionar ao carrinho, ir ao checkout, preencher todos seus dados, escolher o meio de pagamento 1 [em seguida, meio de pagamento 2,3, …], forma de pagamento [débito a vista, crédito à vista, crédito parcelado, quantidade de parcelas], informar o número do cartão, DV, Nome como está no cartão, CPF… cansa até para descrever, quem dirá testar!

Todo o fluxo descrito acima pode ser automatizado e executado com apenas um comando, na verdade semi automatizado pois ainda precisa de sua intervenção, mas ela é muito simples e rápida.

Quais são os passos do TDD?

O TDD (Test Driven Development) baseia-se em três passos, vermelho-verde-refatora. O vermelho é a escrita do primeiro teste antes mesmo da lógica existir. O verde é o ponto em que a lógica para que o teste previamente criado passe. Esta lógica deve ser desenvolvida da forma mais simples possível eliminando complexidades desnecessárias fazendo com que a evolução do código ocorra de forma segura. O refatora é a melhoria do código. Neste ponto são removidas duplicações, múltiplas responsabilidades e o código fica cada vez mais próximo de sua versão final.

Para que o processo vermelho-verde-refatora seja de fato implementado, utiliza-se baby steps ou passos de bebê. Esta técnica consiste em realizar um pequeno passo de cada vez, se uma lógica é complexa de ser desenvolvida ela é dividida em muitas pequenas partes que evoluem até sua solução final. Engana-se quem pensa que baby steps é sair adicionando linha, teste, outra linha, teste, outra linha… Baby steps resume-se em criar a lógica mínima necessária para que o teste passe, ou seja, se um método de teste espera ter como retorno do método testado um array contendo no mínimo 1 registro de usuário, tal método testado (que é nosso código de produção) deve realizar esta lógica de forma minimalista para que seja atendida. Após o primeiro teste passar, começam-se adicionar suas condicionais se as mesmas se fizerem mnecessárias. Com isso, ao realizar a busca de usuários que nasceram no ano de 1986 a lógica mínima que caracteriza-se

como um baby step é o método realizar uma consulta al banco de dados com a condicional da data desejada e retornando um array com todos os usuário localizados.

Baby Steps

Baby Step é o mínimo passo necesário para que o teste passe adequadamente. Pense no cenário de uma calculadora, a soma de 2 + 2 deve ser igual a 4. Nele o teste escrito deve garantir que o valor 4 é retornado, já no método que fará o teste passar o passo mais básico para tal asserção é retornar imediatamente o valor 4, sem realizar calculo nem verificar se recebeu os parâmetros corretos. Com isso o teste passa. O próximo passo é garantir que o método está recebendo os parâmetros mínimos necessários, após isto finalmente o retorno do calculo esperado.

Perceba que foram 3 ações, ou seja, de pouco em pouco o código chegou a sua versão definitiva neste momento. A qualquer momento o mesmo pode sofrer alteração, isso quase sempre ocorre e tem uma chance enorme de continuar ocorrendo pois o software tende a sempre evoluir.

FIRST

O ideal é que ao trabalharmos com TDD utilizemos os princípios F.I.R.S.T.

F: Fast - Os testes precisam ser executados de forma rápida, muito rápida;
I: Isolates - Devem ser isolados, ou seja, testar somente UMA unidade, mesmo sendo difícil, NENHUM fator externo pode interferir no resultado do teste de unidade;
R: Repeteable - Devem ser executáveis em qualquer ordem e com qualquer valor e sempre devem funcionar;
S: Self-Validating - Devem ser auto-suficientes, ou seja, você não deve intervir manualmente para que o mesmo passe;
T: Timely - ao utilizar TDD os testes DEVEM vir antes do código de produção.

Capítulo 2

Iniciando a prática do TDD.

A partir de agora este documento terá uma abordagem mais prática. Apenas precisamos ter conhecimento da ferramenta que utilizaremos, o PHPUnit.

O que é PHPUnit

PHPUnit é um framework de testes em PHP, faz parte da família xUnit que contempla frameworks de testes em diversas linguagens como JUnit no Java, qUnit no javascript entre outras.

Criado e mantido por Sebastian Bergmann está em sua versão 4 e evoluindo constantemente. É de longe o mais utilizado com o PHP. Outra alternativa ao PHPUnit é o SimpleTest, mas não falaremos sobre ele.

Instalando o PHPUnit

Iremos instalar, configurar e utilizar o PHPUnit através do gerenciador de dependências composer. Se você não conhece o composer sugiro a leitura de Composer para iniciantes - Tableless.

Primeiramente crie um arquivo chamando-o de *composer.json*. Nele definiremos o nome de nosso projeto e suas dependências.

```
{
    "name": "Finanças Pessoais",
    "description": "Uma API para gerenciamento de finanças pessoais",
    "authors": [
        {
            "name": "Andre Cardoso",
            "email": "andrecardosodev@gmail.com"
        }
    ],
    "require": {
        "php": ">=5.4"
    },
    "require-dev" : {
        "phpunit/phpunit" : "4.*"
    }
}
```

No arquivo acima definimos de forma muito básica e simplificada o nome de nossa aplicação, sua descrição, detalhes do autor, dependências em modo de produção (“require”) e dependências em modo de desenvolvimento (“require-dev”). Perceba que como executaremos testes apenas em ambiente de desenvolvimento o correto é sempre informarmos o PHPUnit dentro de “require-dev”.

Agora que já temos nossas dependências declaradas basta que baixemos o composer através de um dos métodos abaixo:

cURL

```
$ curl -sS https://getcomposer.org/installer | php
```

ou

PHP

```
$ php -r "readfile('https://getcomposer.org/installer');" | php
```

Os comandos acima baixam o composer e o deixam pronto para uso na raiz de nossa aplicação. Nossa estrutura inicial é a seguinte:

```
├── /src
|   ├── /FinancaPessoal
├── composer.json
├── composer.phar
```

Perceba que nossa aplicação inicia-se em **src**, isto para seguir boas práticas recomendadas pela FIG. A pasta *FinancaPessoal* será o namespace de nossa aplicação.

Como já possuímos as declarações de nossas dependências bem como o próprio composer, basta que rodemos o comando **php composer.phar install** e o resultado será semelhante ao listado abaixo.

```
$ php composer.phar install
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing sebastian/version (1.0.3)
    Downloading: 100%

  - Installing sebastian/exporter (1.0.1)
    Downloading: 100%

  - Installing sebastian/environment (1.0.0)
    Downloading: 100%

  - Installing sebastian/diff (1.1.0)
    Downloading: 100%

  - Installing symfony/yaml (v2.4.2)
    Downloading: 100%

  - Installing phpunit/php-text-template (1.2.0)
    Downloading: 100%

  - Installing phpunit/phpunit-mock-objects (2.0.4)
    Downloading: 100%

  - Installing phpunit/php-timer (1.0.5)
    Downloading: 100%

  - Installing phpunit/php-token-stream (1.2.2)
    Downloading: 100%
```

```
  - Installing phpunit/php-file-iterator (1.3.4)
    Downloading: 100%

  - Installing phpunit/php-code-coverage (2.0.4)
    Downloading: 100%

  - Installing phpunit/phpunit (4.0.13)
    Downloading: 100%

phpunit/phpunit suggests installing phpunit/php-invoker (~1.1)
Writing lock file
Generating autoload files
```

Feito isto já temos o PHPUnit devidamente instalado juntamente com todas suas dependências. Nossa nova estrutura de pastas é a seguinte:

```
├── /src
|   ├── /FinancaPessoal
├── /vendor
|   ├── /bin
|   ├── /composer
|   ├── /phpunit
|   ├── /sebastian
|   ├── /symphony
|   ├── autoload.php
├── composer.json
├── composer.lock
├── composer.phar
```

- O arquivo *src/composer.lock* possui as informações do estado atual de cada um dos pacotes instalados. A partir dele é verificada e baixada atualização através do update do composer quando necessário.
- O arquivo *src/vendor/autoload.php* é responsável por registrar todos os namespaces de cada pacote que estamos utilizando e deve ser chamado em seu arquivo de inicialização da aplicação para que ao rodar a aplicação todos os namespaces estejam devidamente carregados.

Se rodarmos o comando **./vendor/bin/phpunit** já veremos o menu de ajuda dp PHPUnit.

```
$ ./vendor/bin/phpunit
PHPUnit 4.0.13 by Sebastian Bergmann.

Usage: phpunit [options] UnitTest [UnitTest.php]
       phpunit [options] <directory>

Code Coverage Options:

  --coverage-clover <file>  Generate code coverage report in Clover XML format
```

```
.
  --coverage-crap4j <file>  Generate code coverage report in Crap4J XML format
.
  --coverage-html <dir>     Generate code coverage report in HTML format.
  --coverage-php <file>     Export PHP_CodeCoverage object to file.
  --coverage-text=<file>    Generate code coverage report in text format.
                            Default: Standard output.
  --coverage-xml <dir>      Generate code coverage report in PHPUnit XML forma
t.
....
```

Primeiro teste

Agora será criado nosso primeiro caso de teste. Para isto teremos de criar uma nova pasta em nossa aplicação chamando-a de *tests* dentro dela uma pasta chamada *src* e dentro de src uma pasta chamada *FinancaPessoal*. Isto serve para respeitarmos o namespace de nossa aplicação. Desta forma tanto o código de produção quanto o de testes são utilizáveis a partir de `php use FinancaPessoal`. Dentro da pasta *FinancaPessoal* de tests crie uma pasta chamada *Filter* e dentro dela um arquivo *FormataMoedaTest.php*. Resumidamente crie o arquivo ***tests/src/FinancaPessoal/Filter/FormataMoedaTest.php***. Nossa nova estrutura de pastas é:

```
├── /src
|   ├── /FinancaPessoal
├── /tests
|   ├── /src
|       ├── /FinancaPessoal
|           ├── /Filter
|               ├── FormataMoedaTest.php
├── /vendor
|   ├── /bin
|   ├── /composer
|   ├── /phpunit
|   ├── /sebastian
|   ├── /symphony
|   ├── autoload.php
├── composer.json
├── composer.lock
├── composer.phar
```

No arquivo FormataMoedaTest.php adicione o seguinte conteúdo:

```php
<?php
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

class FormataMoedaTest extends PHPUnit_Framework_TestCase
{

    protected $formataMoeda = null;
```

```
    public function setUp()
    {
        parent::setUp();
    }

    public function tearDown()
    {
        parent::tearDown();
    }

}
```

Como é possível perceber, nosso teste deve estender da classe *PHPUnit_Framework_TestCase*. Os métodos que ali se encontram, `setUp()` e `tearDown()` já existem na classe do PHPUnit mas você pode utilizá-los para construir e destruir respectivamente, objetos, arquivos, conexões e qualquer outra coisa que seja necessário entre um teste e outro.

> A classe de teste deve seguir a convenção de nomenclatura que é:
>
> **Nome de classe:** NomeDesejado seguido de Test (NomeDesejadoTest)
>
> **Nome de métodos de teste:** test seguido de uma descrição o mais clara possível sobre o que o teste vai garantir que está funcionando (testClassExists, testUserHasEmail e assim por diante)

Funcionamento básico de um teste

O teste sempre começa com o setUp onde uma sequência de funcionalidades necessárias para que o teste rode minimamente são inicializadas. Em seguida o teste é executado e por último o tearDown desfaz o que o teste fez para que não fique vestígios ou sobras antes de partir para o próximo teste. Pense num cenário onde temos três testes, a sequência é:

```
setUp > teste 1 > tearDown
setUp > teste 2 > tearDown
setUp > teste 3 > tearDown
```

Neste exemplo apenas inicializaremos nossa classe de produção que será testada e no tearDown simplesmente a destruiremos.

```
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

public function setUp()
{
    parent::setUp();
    $this->formataMoeda = new \FinancaPessoal\Filter\FormataMoeda();
```

```
    }

    public function tearDown()
    {
        unset($this->formataMoeda);
        parent::tearDown();
    }
```

Em seguida o primeiro teste que comumente é executado é se a classe testada existe.

```
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

public function testIfClassExists()
{
    $this->assertTrue(class_exists('FinancaPessoal\Filter\FormataMoeda'));
}
```

Neste teste estou garantindo que a classe existe através do **assertTrue** da função **class_exists**.

Se rodarmos nosso primeiro teste com o comando **./vendor/bin/phpunit tests/src/FinancaPessoa/Filter/FormataMoedaTest** veremos alguns erros conforme mostra abaixo:

```
$ ./vendor/bin/phpunit tests/src/FinancaPessoal/Filter/FormataMoedaTest
PHPUnit 4.0.13 by Sebastian Bergmann.

PHP Fatal error:  Class 'FinancaPessoal\Filter\FormataMoeda' not found in /hom
e/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Filter/FormataMoeda.ph
p on line 11
```

Este erro era realmente esperado pois ainda não temos nossa classe testada dentro de *src/FinancaPessoal/Filter*. Criaremos a mesma agora.

```
// src/FinancaPessoal/Filter/FormataMoeda.php

namespace FinancaPessoal\Filter;

class FormataMoeda
{

}
```

Se rodarmos novamente o comando **./vendor/bin/phpunit tests/src/FinancaPessoa/Filter/FormataMoedaTest** veremos que o erro ainda existe. Isto porque não informamos onde encontra-se nosso namespace e nem utilizamos o autoload para que o mesmo esteja disponível para uso. Faremos isso

agora.

Crie na raiz de sua aplicação um arquivo chamado *bootstrap.php* populando-o com o seguinte conteúdo:

```
// bootstrap.php

//AutoLoader do Composer
$loader = require __DIR__.'/vendor/autoload.php';

//informando o namespace  e sua localização
$loader->add('FinancaPessoal', __DIR__ . '/src');
```

E agora em nossa classe de testes apenas damos um require no bootstrap. Lembrando que estamos sempre executando o PHPUnit da raiz da aplicação então o require pode ser sem alteração de diretórios como mostra abaixo.

```
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

require 'bootstrap.php';

class FormataMoedaTest extends PHPUnit_Framework_TestCase
...
```

Rodando o teste novamente temos o resultado finalmente esperado, a classe realmente existe!

```
$ ./vendor/bin/phpunit tests/src/FinancaPessoal/Filter/FormataMoedaTest
PHPUnit 4.0.13 by Sebastian Bergmann.

.

Time: 37 ms, Memory: 2.00Mb

OK (1 test, 1 assertion)
```

Um novo teste que realizamos é garantir que o método *brlParaFloat* existe. Este método limpará a formatação do valor em Reais para um simples float que será gravado no banco.

```
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

public function testIfMethodBrlParaFloatExists()
{
    $this->assertTrue(method_exists($this->formataMoeda, 'brlParaFloat'));
}
```

Obviamente que se rodarmos o teste o mesmo deve quebrar pois o método ainda

não existe em nossa classe testada.

```
$ ./vendor/bin/phpunit tests/src/FinancaPessoal/Filter/FormataMoedaTest
PHPUnit 4.0.13 by Sebastian Bergmann.

.F

Time: 38 ms, Memory: 2.00Mb

There was 1 failure:

1) FormataMoedaTest::testIfMethodBrlParaFloatExists
Failed asserting that false is true.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Filter/FormataMoed
aTest.php:29

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Com isso criamos nosso método na classe testada e o teste passa.

```


/**
*
* @param string $valor
* @return float
*/
public function brlParaFloat( $valor )
{

}
```

```
$ ./vendor/bin/phpunit tests/src/FinancaPessoal/Filter/FormataMoedaTest
PHPUnit 4.0.13 by Sebastian Bergmann.

..

Time: 37 ms, Memory: 2.00Mb

OK (2 tests, 2 assertions)
```

O PHPUnit possui vários asserts e com toda certeza ao longo do workshop não veremos nem a metade deles, resta a você se aprofundar no desenvolvimento orientado a testes e utilizar cada qual conforme sua necessidade.

Apenas para fechar esta seção, vamos fazer o método *brlParaFloat* realizar sua funcionalidade como desejado.

```php
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

public function testFilterBrlParaFloat()
{
    $valorEmReais = 'R$ 23,95';

    $result = $this->formataMoeda->brlParaFloat($valorEmReais);

    $this->assertNotNull($result);
    $this->assertInternalType('float', $result);
    $this->assertEquals(23.95, $result);
}
```

```php
// src/FinancaPessoal/Filter/FormataMoeda.php

/**
*
* @param string $valor
* @return float
*/
public function brlParaFloat( $valor )
{
    $find = array('R$ ', '.', ',');
    $replace = array('', '', '.');

    return (float) str_replace($find, $replace, $valor);
}
```

Feito isto nosso teste passa.

```
$ ./vendor/bin/phpunit tests/src/FinancaPessoal/Filter/FormataMoedaTest
PHPUnit 4.0.13 by Sebastian Bergmann.

...

Time: 43 ms, Memory: 2.25Mb

OK (3 tests, 5 assertions)
```

Da forma como os testes estão sendo executados no momento, somente os presentes na classe *FormataMoedaTest* que rodam. Caso existam testes em alguma outra classe os mesmos ainda não serão abordados.

Capítulo 3

Testando persistência de dados.

Neste capítulo trabalharemos com persistência de dados através do ORM Doctrine.

Baixando o ORM Doctrine

Para persistência ao banco de dados utilizaremos o ORM Doctrine. Pegando emprestada a descrição do próprio site “O Projeto Doctrine é o lar de várias bibliotecas PHP focadas principalmente no armazenamento de banco de dados e mapeamento de objetos.”

Assim como o próprio PHPUnit, instalaremos o Doctrine através do composer. O mesmo será adicionado em “require” de nosso arquivo *composer.json* e sua nova estrutura será:

```
    "require": {
        "php": ">=5.4",
        "doctrine/orm" : "2.4.*"
    },
```

Agora temos de rodar o comando **php composer.phar update** na raíz de nosso projeto.

```
$ php composer.phar update
Loading composer repositories with package information
Updating dependencies (including require-dev)
    - Installing symfony/console (v2.4.2)
      Loading from cache

  - Installing doctrine/lexer (v1.0)
    Loading from cache

  - Installing doctrine/annotations (v1.1.2)
    Loading from cache

  - Installing doctrine/collections (v1.2)
    Loading from cache

  - Installing doctrine/cache (v1.3.0)
    Loading from cache

  - Installing doctrine/inflector (v1.0)
    Loading from cache

  - Installing doctrine/common (v2.4.1)
    Loading from cache

  - Installing doctrine/dbal (v2.4.2)
    Loading from cache

  - Installing doctrine/orm (v2.4.2)
    Loading from cache
```

```
symfony/console suggests installing symfony/event-dispatcher ()
Writing lock file
Generating autoload files
```

Como é possível ver, tudo que se faz necessário para o correto funcionamento do Doctrine foi instalado, mesmo sendo definido apenas o Doctrine em sua versão 2.4. Feito isto o Doctrine já está disponível para utilizarmos restando apenas sua correta configuração.

Criando o boostrap da aplicação

O bootstrap de nossa aplicação na verdade já existe na raiz da mesma. Apenas adicionaremos as configurações necessárias para o funcionamento do Doctrine. Basicamente precisamos informar os dados da conexão com o banco de dados, onde nossas entidades serão carregadas e como serão criadas. Entende-se como entidade um objeto, uma classe que conterá as definições de cada uma das tabelas de nosso banco de dados bem como seus possíveis relacionamentos. Cada tabela no banco de dados é representado por uma entidade.

Segue a nova estrutura de nosso arquivo bootstrap.php.

```php
<?php
// bootstrap.php

//AutoLoader do Composer
$loader = require __DIR__.'/vendor/autoload.php';

//informando o namespace  e sua localização
$loader->add('FinancaPessoal', __DIR__ . '/src');

// indicando tudo que usaremos no bootstrap
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;
use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\Common\Annotations\AnnotationRegistry;


/**
 * Definindo se é modo desenvolvimento
 *
 * Caso true: o cache do Doctrine é realizado em formato de array
 * Caso false: o cache é conforme configuração (memcache, APC..)
 *
 * Somente trabalharemos aqui com o modo TRUE, cache em array
 */
$isDevMode = true;
$config = Setup::createConfiguration( $isDevMode );
```

```php
// pasta onde encontra-se nossas entidades
$entitypath = array( __DIR__ . '/src/FinancaPessoal/Entity' );


// registrando as entidades
$driver = new AnnotationDriver(new AnnotationReader(), $entitypath);
$config->setMetadataDriverImpl($driver);

/**
 * indicando que trabalharemos com o modo annotations para
 * as entidades. Pode ser também via arquivo yaml e xml
 *
 */
AnnotationRegistry::registerFile(__DIR__ . '/vendor/doctrine/orm/lib/Doctrine/
ORM/Mapping/Driver/DoctrineAnnotations.php');


// configurando a conexão com o banco de dados
$conn = array(
    'driver' => 'pdo_mysql',
    'host' => 'localhost',
    'port' => 3306,
    'user' => 'root',
    'password' => 'root',
    'dbname' => 'financas',
);

// E finalmente criando o manipulador de entidades
$entityManager = EntityManager::create($conn, $config);
```

Para utilizar o Doctrine em qualquer ponto de nossa aplicação a partir de agora basta chamar a variável `$entityManager`.

Agora que temos a configuração de nossa aplicação em modo de produção precisamos criar a conexão para a realização dos testes. Dentro de *tests* crie um arquivo chamado *bootstrap.php* e nele adicione o seguinte conteúdo:

```php
<?php
// tests/bootstrap.php

require dirname(__DIR__) . '/bootstrap.php';

use Doctrine\ORM\EntityManager;


$conn = array(
    'driver' => 'pdo_sqlite',
    'path' => ':memory:',
);

// obtaining the entity manager
$entityManager = EntityManager::create($conn, $config);
```

```
return $entityManager;
```

No bootstrap de testes estamos apenas utilizando tudo que o bootstrap de produção já nos fornece e apenas sobrescrevemos a conexão com o banco de dados. Em modo de produção o banco conectado foi o mysql, nos testes utilizaremos o banco sqlite em memória para que os testes rodem de forma muito mais rápida.

Ok, até aqui já temos as configurações necessárias para a execução dos testes e de nossa aplicação em modo de produção, mas detalhe que são apenas configurações pois nada ainda está funcionando de fato. Criaremos os testes necessários para cada uma de nossas entidades e em sequência a cada teste criaremos o necessário para que o mesmo passe.

Xml de configurações

Para que nossos teste fluam de forma mais simples criaremos um arquivo xml com algumas configurações básicas. Neste arquivo definiremos como e onde serão realizados os testes bem como o que precisamos ignorar ao executar cada um dos testes. Basicamente ignoramos tudo que estiver em *vendor* que foram instalados pelo composer, os pacotes que ali se encontram JÁ foram testados pelos seus fornecedores. Dentro de *tests* crie um arquivo chamado *phpunit.xml* com o conteúdo a seguir.

```
<!-- phpunit.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!-- O atributo colors serve para que tenhamos um visual mais fácil de entende
r -->

<!-- O atributo bootstrap carrega as configurações iniciais -->
<phpunit colors="true" bootstrap="bootstrap.php">

    <!-- Indicando qual é o diretório onde as classes de teste se encontram --
>
    <testsuites>
        <testsuite name="FinancaPessoal Suite">
            <directory suffix=".php">src/</directory>
        </testsuite>
    </testsuites>

<!-- Adicionando filtros, basicamente whitelist (diretórios que serão executad
os),
    dentro temos o exclude (diretórios que não serão executados pelos testes)
-->
    <filter>
        <whitelist>
            <directory suffix=".php">../src/</directory>
            <exclude>
                <directory suffix=".php">./vendor/</directory>
```

```
            </exclude>
        </whitelist>
    </filter>
</phpunit>
```

Com o arquivo xml em mãos basta que rodemos o comando **./vendor/bin/phpunit -c tests/phpunit.xml**.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.....

Time: 102 ms, Memory: 4.25Mb

OK (5 tests, 9 assertions)
```

> O parâmetro **-c** serve para carregar um arquivo de configuração.

Criando o TestCase

Para simplificarmos as coisas criaremos agora uma classe chamada *TestCase*. Nesta classe definiremos algumas configurações que serão executadas comumene em todos os testes como criação/destruição de tabelas em nosso banco de dados de teste, configuração do entityManager entre outras possibilidades.

Crie dentro da pasta *src* uma pasta chamada *Test* e dentro dela um arquivo chamado *TestCase.php* com o seguinte conteúdo:

```
<?php
// src/Test/TestCase.php

namespace FinancaPessoal\Test;

use PHPUnit_Framework_TestCase as PHPUnit;

class TestCase extends PHPUnit
{

    public function setUp()
    {
        parent::setUp();
    }

    public function tearDown()
```

```
    {
        parent::tearDown();
    }

}
```

Como você pôde perceber, esta classe está bem enxuta, ainda. Isto porque futuramente adicionaremos configurações de banco de dados, criação e remoção de tabelas no banco de dados e alguns atributos padrões.

Nossa já existente classe de teste (FormataMoedaTest) já não mais necessita do require de bootstrap adicionado no início de nosso teste e agora deve ser definido seu namespace e utilizar a classe *TestCase*. FormataMoedaTest agora estenderá de *TestCase* ao invés de *PHPUnit_Framework_TestCase* como estava anteriormente.

> Toda e qualquer nova classe de testes agora terá a definicção do namespace, utilizará e estenderá de *FinancaPessoa\Test\TestCase*.

```
// tests/src/FinancaPessoal/Filter/FormataMoedaTest.php

// antes
require 'bootstrap.php';

class FormataMoedaTest extends PHPUnit_Framework_TestCase


// depois
namespace FinancaPessoal\Filter;

use FinancaPessoal\Test\TestCase;

class FormataMoedaTest extends TestCase
```

Criando e testando entidades

Seguindo o conceito de desenvolvimento orientado a testes, obviamente que começaremos pelos testes. Dentro de *tests/src/FinancaPessoal* crie uma pasta chamada *Entity* e dentro dela um arquivo chamado *UserTest.php*, nele adicione o seguinte conteúdo.

```
<?php
// tests/src/FinancaPessoal/Entity/UserTest.php

namespace FinancaPessoal\Entity;

use FinancaPessoal\Test\TestCase;
```

```
class UserTest extends TestCase
{

    public function testClassExists()
    {
        $this->assertTrue(class_exists('FinancaPessoal\Entity\User'));
    }

}
```

Ao rodarmos o teste (**./vendor/bin/phpunit -c tests/únit.xml**) o mesmo simplesmente quebrará. Isto porque ainda não existe a classe testada.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

F.....

Time: 139 ms, Memory: 4.50Mb

There was 1 failure:

1) FinancaPessoal\Entity\UserTest::testClassExists
Failed asserting that false is true.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Entity/UserTest.ph
p:12

FAILURES!
Tests: 6, Assertions: 10, Failures: 1.
```

Apenas criando a classe *src/FinancaPessoal/Entity/User* o teste já passa.

```
<?php
// src/FinancaPessoal/Entity/User.php

namespace FinancaPessoal\Entity;


class User{}
```

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

......
```

```
Time: 104 ms, Memory: 4.50Mb

OK (6 tests, 10 assertions)
```

Agora verificaremos se todos os métodos que esperamos que a classe possua já estão criados através de um teste para cada. Mas primeiro criaremos dois atributos que serão utilizados várias vezes em nossos testes, o *entityName* e também o *entity* que são o nome da entidade testada e a instância da mesma, respectivamente. Os métodos setUp e tearDown monta e destroi a entidade, conforme já explicado anteriormente.

```php
// tests/src/FinancaPessoal/Entity/UserTest.php

    protected $entityName;
    protected $entity;

    public function setUp()
    {
        parent::setUp();
        $this->entityName = 'FinancaPessoal\Entity\User';
        $this->entity = new $this->entityName();
    }

    public function tearDown()
    {
        unset($this->entityName, $this->entity);
        parent::tearDown();
    }
```

Com isso podemos também alterar nosso teste que verifica se a classe existe.

```php
// tests/src/FinancaPessoal/Entity/UserTest.php

// antes
public function testClassExists()
{
    $this->assertTrue(class_exists('FinancaPessoal\Entity\User'));
}

// depois
public function testClassExists()
{
    $this->assertTrue(class_exists($this->entityName));
}
```

Antes de criarmos qualquer teste, criaremos os atributos de nossa entidade *User*. Não serão explicadas as notações do Doctrine por não ser este o foco deste material.

```php
<?php
// src/FinancaPessoal/Entity/User.php

namespace FinancaPessoal\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * User
 *
 * @ORM\Table(name="users")
 * @ORM\Entity
 */
class User
{

    /**
     * @ORM\Column(type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue
     * @var int
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255, nullable=false)
     * @var string
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=false)
     * @var string
     */
    private $email;

    /**
     * @ORM\Column(type="string", length=255)
     * @var string
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=500, nullable=true)
     * @var string
     */
    private $avatar;

}
```

Nossos novos testes (verificando se os métodos desejados existem) serão criados um em seguida do outro e toda lógica necessária para que o teste passe será construída em seguida ao teste.

```
// tests/src/FinancaPessoal/Entity/UserTest.php

public function testIfMethodToArrayExists()
{
    $this->assertTrue(method_exists($this->entity, 'toArray'));
}
```

Para que este teste passe, nosso baby-step deve passar de forma mínima, ou seja, apenas criando o método na entidade *User*, perceba que o teste está apenas garantindo que o método existe, não que algum comportamento é esperado dele.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.F.....

Time: 107 ms, Memory: 4.50Mb

There was 1 failure:

1) FinancaPessoal\Entity\UserTest::testIfMethodToArrayExists
Failed asserting that false is true.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Entity/UserTest.ph
p:32

FAILURES!
Tests: 7, Assertions: 11, Failures: 1.
```

Criamos o mesmo e o teste passa.

```
// src/FinancaPessoal/Entity/User.php

public function toArray(){}
```

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.......

Time: 105 ms, Memory: 4.50Mb

OK (7 tests, 11 assertions)
```

O próximo teste consiste em testar se o comportamento do método *toArray*. Este deve retornar os dados do Usuário em formato de array. Para isso utilizaremos uma biblioteca do Zend Framework 2 que realiza tal tratamento, o Hydrator.

```
// tests/src/FinancaPessoal/UserTest.php

public function testIfToArrayIsReturningAnArray()
{
    $this->assertInternalType('array', $this->entity->toArray());
}
```

Rodamos o teste e ele quebra como esperado:

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

..F.....

Time: 110 ms, Memory: 4.50Mb

There was 1 failure:

1) FinancaPessoal\Entity\UserTest::testIfToArrayIsReturningAnArray
Failed asserting that null is of type "array".

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Entity/UserTest.ph
p:37

FAILURES!
Tests: 8, Assertions: 12, Failures: 1.
```

Para que este teste passe precisamos do Hydrator do Zend, o qual faz parte da biblioteca stdlib do mesmo e definiremos em nosso arquivo *composer.json*. Também faz-se necessária a instalação de Zend Filter se Service Manager.

```
"require": {
    ...
    "zendframework/zend-stdlib": "2.3.*@dev",
    "zendframework/zend-filter": "2.3.*@dev",
    "zendframework/zend-servicemanager": "2.3.*@dev"
},
```

Feito isto faz-se necessário o update do composer:

```
$ php composer.phar update
Loading composer repositories with package information
Updating dependencies (including require-dev)
```

```
  - Installing zendframework/zend-stdlib (dev-develop 3acce3d)
    Cloning 3acce3d524c5c5b7e92c90b0c2763068c1efaf2c

  - Installing zendframework/zend-filter (dev-develop c878eb7)
    Cloning c878eb757c53057a9cd2c7af391eec69ab4bca61

  - Installing zendframework/zend-servicemanager (dev-develop 4911c6c)
    Cloning 4911c6c6f58d7058e351e6fe8ecff14666da1259

zendframework/zend-stdlib suggests installing zendframework/zend-eventmanager
(To support aggregate hydrator usage)
zendframework/zend-stdlib suggests installing zendframework/zend-serializer (Z
end\Serializer component)
Writing lock file
Generating autoload files
```

Agora que já temos o Stdlib podemos utilizar o Hydrator em nossa entidade User.

```
<?php
// src/FinancaPessoal/Entity/User.php

...
use Zend\Stdlib\Hydrator;
...

/**
 *
 * @return array
*/
public function toArray()
{
    $hydrator = new Hydrator\ClassMethods();
    return $hydrator->extract($this);
}
```

E agora o teste que garante que o método *toArray* realmente está retornando os dados em formato de array passa.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

........

Time: 115 ms, Memory: 4.75Mb

OK (8 tests, 12 assertions)
```

Mas somente estes testes em *toArray* não são suficiente, estamos garantindo que

ele existe e que está retornando um array, mas será que este array realmente possui os dados do usuário?

```
// tests/src/FinancaPessoal/Entity/UserTest.php

public function testIfToArrayIsReturningValidArray()
{
    $result = $this->entity->toArray();
    $this->assertNotEmpty($result);
    $this->assertArrayHasKey('name', $result);
    $this->assertArrayHasKey('email', $result);
    $this->assertArrayHasKey('password', $result);
}
```

Obviamente que o teste quebra pois o array retornado está vazio. Para que este teste passe, precisamos criar todos os getters de nossa entidade.

```
// src/FinancaPessoal/Entity/User.php

...
public function getId()
{
    return $this->id;
}

public function getName()
{
    return $this->name;
}

public function getEmail()
{
    return $this->email;
}

public function getPassword()
{
    return $this->password;
}

public function getAvatar()
{
    return $this->avatar;
}
...
```

Agora o teste passa corretamente e vemos que nosso método *toArray* existe, está retornando um array VÁLIDO.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
```

```
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.........

Time: 128 ms, Memory: 5.00Mb

OK (9 tests, 16 assertions)
```

Agora definiremos a senha de um usuário e garantiremos que a mesma não será gravada como string pura no banco de dados. Como estamos desenvolvendo orientado a testes precisamos primeiro criar nosso método de teste.

```
// tests/src/FinancaPessoal/Entity/UserTest.php

public functionn testIfPasswordHashWasApplied()
{
    $password = 'senha em plain text';
    $user = new \FinancaPessoal\Entity\User();

    $user->setPassword($password);

    $this->assertNotEquals($password, $user->getPassword());
}
```

Obviamente como esperado o teste quebra. Isto porque não definimos nenhum hash para a senha, com isso da mesma forma que ela é setada seria gravada no banco. Temos de corrigir isto. Esta correção nos custará criar um salt que será gerado dinamicamente. Este salt serve para que ao fornecer a senha de login por exemplo o mesmo seja combinado com tal senha, aplicado o hash e verificado se a senha que o usuário digitou confere com a que foi gravada no banco de dados. Alguns programadores (e frameworks) utilizam salt único para a aplicação, o que não está errado, é apenas uma outra abordagem.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

....F.....

Time: 551 ms, Memory: 5.00Mb

There was 1 failure:

1) FinancaPessoal\Entity\UserTest::testIfPasswordHashWasApplied
Failed asserting that 'senha em plain text' is not equal to <string:senha em p
lain text>.
```

```
/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Entity/UserTest.ph
p:56

FAILURES!
Tests: 10, Assertions: 17, Failures: 1.
```

Para que nossa senha não seja gravada de forma texto puro no banco de dados criaremos em nossa entidade o atributo salt. No construtor de nossa entidade geraremos um salt aleatório. Este salt será unido à senha fornecida pelo usuário no momento de um futuro login (que não sera visto aqui), será realizado o hash desta combinação que caracterizar-se-a como a senha que será gravada no banco.

Serão necessárias mais dois componentes do Zend Framework, o Math e o Crypt. Ambos deverão ser registrados no “require” de nosso composer.json.

```
"require": {
        "php": ">=5.4",
        "doctrine/orm" : "2.4.*",
        "zendframework/zend-stdlib": "2.3.*@dev",
        "zendframework/zend-filter": "2.3.*@dev",
        "zendframework/zend-servicemanager": "2.3.*@dev",

        # novos pacotes necessários
        "zendframework/zend-math": "2.3.*@dev",
        "zendframework/zend-crypt": "2.3.*@dev"
    },
```

Após esta declaração rode o comando **php composer.phar update**.

Agora resta em nossa entidade definir o hash e o setPassword para que nosso teste passe.

```
// src/FinancaPessoal/Entity/User.php

// novos uses
use Zend\Math\Rand;
use Zend\Crypt\Key\Derivation\Pbkdf2;


// Adicione a nova propriedade salt
/**
* @ORM\Column(type="text")
* @var string
*/
private $salt;


// crie o construtor
public function __construct()
```

```
{
    $this->salt = base64_encode(Rand::getBytes(8, true));
}

// e finalmente crie um método chamado setPassword

/**
*
* @param string $password
* @return \FinancaPessoal\Entity\User
*/
public function setPassword($password)
{
    $this->password = base64_encode(Pbkdf2::calc('sha256', $password, $this->s
alt, 10000, strlen($password*2)));

    return $this;
}
```

Agora rodamos nossos testes novamente e tudo passa como esperávamos.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.........

Time: 205 ms, Memory: 5.00Mb

OK (10 tests, 17 assertions)
```

Para que possamos finalizar de vez nossos testes na entidade *User* apenas vamos setar os dados e verificar se cada um deles está correto, isto basicamente não se faz necessário mas será mostrado aqui somente para que fique claro que a classe toda de fato foi testada.

```
// tests/src/FinancaPessoal/Entity/UserTest.php

public function testSetDataAndverifyIntegrity()
{
    $data = array(
        'name' => 'Seu nome',
        'email' => 'seuemail@domain.com',
        'password' => '12345678',
        'avatar' => 'imagem.png'
    );

    $user = new \FinancaPessoal\Entity\User( $data );

    $this->assertNotNull($user);
```

```
        $this->assertEquals('Seu nome', $user->getName());
        $this->assertEquals('seuemail@domain.com', $user->getEmail());
        $this->assertNotEquals('12345678', $user->getPassword());
        $this->assertEquals('imagem.png', $user->getAvatar());
    }
```

Se rodarmos os testes agora veremos que o mesmo falhará. Isto porque estamos setando os atributos de user de forma diferente da esperada. Atualmente nosso User possui apenas o setter de password, criaremos dos demais atributos com excessão do salt que é gerado no construtor da entidade, do password que já existe com a configuração desejada.

```
// src/FinancaPessoal/Entity/User.php

public function setId($id)
{
    $this->id = $id;
    return $this;
}

public function setName($name)
{
    $this->name = $name;
    return $this;
}

public function setEmail($email)
{
    $this->email = $email;
    return $this;
}

public function setAvatar($avatar)
{
    $this->avatar = $avatar;
    return $this;
}
```

Mas... se rodarmos os testes os mesmos ainda não passam...

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.....F.....

Time: 210 ms, Memory: 5.25Mb

There was 1 failure:
```

```
1) FinancaPessoal\Entity\UserTest::testSetDataAndverifyIntegrity
Failed asserting that null matches expected 'Seu nome'.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Entity/UserTest.ph
p:71

FAILURES!
Tests: 11, Assertions: 19, Failures: 1.
```

Isto se dá ao fato de que a entidade User esperava receber setName, setEmail, setAvatar... No entanto estamos enviando da seguinte forma:

```
// tests/src/FinancaPessoal/Entity/UserTest.php

$data = array(
            'name' => 'Seu nome',
            'email' => 'seuemail@domain.com',
            'password' => '12345678',
            'avatar' => 'imagem.png'
        );

$user = new \FinancaPessoal\Entity\User( $data );
```

Para que tal teste passe temos de adicionar o hydrate no construtor de FinancaPessoal\Entity\User. Atualizaremos agora o método construtor da classe.

```
// src/FinancaPessoal/Entity/User.php

public function __construct( $data = array() )
{
    $this->salt = base64_encode(Rand::getBytes(8, true));

    // recebendo dados como array e populando atributos da classe
    $hydrator = new Hydrator\ClassMethods();
    $hydrator->hydrate($data, $this);
}
```

E finalmente todos os testes passam novamente.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

...........

Time: 315 ms, Memory: 5.00Mb

OK (11 tests, 22 assertions)
```

Agora que você já conheceu os testes necessários para uma entidade resta criar os testes e entidades que forem necessários para sua aplicação. Não serão criadas as demais entidades neste manual por questão de espaço e também por tornar-se muito repetitivo.

Criando e testando Serviços

Existem muitas formas de realizar ações em nossa aplicação, elas vão desde um Front Controller (que assim como em API's RESTfull agregam todas em um único controller), existe a opção de criarmos controllers específicos para tais ações (assim como em Frameworks MVC como CakePHP, CodeIgniter...) e existe também a possibilidade de criar ações como inserir, editar, ler, deletar como serviços dispostos em nossa aplicação, utilizaremos o conceito desta última para nossa sequencia de testes.

De fato não criaremos tais ações (CRUD) como serviço por não dispormos do nenhum gerenciador de serviços como por exemplo o *serviceManager* do Zend Framework 2 mas sim, criaremos classes seguindo o padrão do Zend.

Ainda possuímos apenas a entidade *User* a qual já realizamos os testes adequados. reforçando que uma entidade é apenas uma representação de uma estrutura de banco de dados, neste caso a entidade *User* representa a tabela *users* em nosso banco de dados.

Nosso primeiro serviço começa obviamente pelos testes. Crie um arquivo chamado *UserTest.php* dentro da pasta `tests/src/FinancaPessoal/Service` e como primeiro teste verifique se a classe `\FinancaPessoal\Service\User` existe.

```php
<?php
// tests/src/FinancaPessoal/Service/UserTest.php

namespace FinancaPessoal\Service;

use FinancaPessoal\Test\TestCase;

class UserTest extends TestCase
{

    protected $serviceName;
    protected $service;

    public function setUp()
    {
        parent::setUp();
        $this->serviceName = '\FinancaPessoal\Service\User';
        $this->service = new $this->serviceName();
    }

    public function tearDown()
    {
```

```
        parent::tearDown();
        unset($this->serviceName, $this->service);
    }

    public function testClassExists()
    {
        $this->assertTrue(class_exists($this->serviceName));
    }

}
```

Ao rodarmos nossos testes espera-se que os mesmos quebrem pois ainda não temos a classe `\FinancaPessoal\Service\User` criada.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

...........PHP Fatal error:  Class '\FinancaPessoal\Service\User' not found in
/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Service/UserTest.p
hp on line 17
```

Relembrando que para que o teste passe precisamos agora criar nossa classe testada.

```
<?php
// src/FinancaPessoal/Service/User.php

namespace FinancaPessoal\Service;

class User
{
    /**
    * @var \Doctrine\ORM\EntityManager
    */
    protected $em;
}
```

Agora o teste passa no entanto ele está garantindo apenas que a classe existe. Adicionaremos a verificação do construtor da classe User.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testMethodConstructExists()
{
    $this->assertTrue(method_exists($this->service, '__construct'));
}
```

Este teste deve falhar, crie o método *__contruct()* em `src/FinancaPessoal/Service/User.php` fazendo o mesmo receber o entityManager do Doctrine.

```
// src/FinancaPessoal/Service/User.php

public function __construct( \Doctrine\ORM\EntityManager $em )
{
    $this->em = $em;
}
```

Ok, agora precisamos em nossa classe de testes injetar a dependência para a classe *\FinancaPessoal\Service\User* que nada mais é nosso entityManager, atualmente o Doctrine. Sobrescreveremos nosso *setUp* em nosso teste de serviços do usuário.

```
// tests/src/FinancaPessoal/ServiceUser.php

public function setUp()
{
    parent::setUp();
    $this->serviceName = '\FinancaPessoal\Service\User';

    // estrutura antiga
    // $this->service = new $this->serviceName();

    // Nova estrutura
    $this->service = new $this->serviceName( $this->em );
}
```

Se rodarmos nossos testes agora todos passam novamente.

> Você pode estar se perguntando, de onde surgiu o $this->em ? Na verdade ele ainda não existe e precisa ser criado na classe pai de \FinancaPessoal\Service\UserTest a classe TestCase.

```
// src/FinancaPessoal/Test/TestCase.php

/**
* @var \Doctrine\ORM\EntityManager $em
*/
protected $em;

public function setUp()
{
    $em = require dirname(dirname(dirname(__DIR__))) . '/tests/bootstrap.php';
    $this->em = $em;
    parent::setUp();
```

```
}
```

Voltando à classe Service\UserTest, agora podemos verificar se o método *save* existe em nossa classe testada.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfMethodSaveExists()
{
    $this->assertTrue(method_exists($this->service, 'save'));
}
```

Aqui o teste falha, faremos o mesmo passar

```
// src/FinancaPessoal/Service/User.php

/**
* @param array $data
*/
public function save( array $data )
{

}
```

Agora falta apenas verificarmos se o método está retornando `\FinancaPessoal\Entity\User` como esperado.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfSaveIsWorkingProperly
{
    $data = array(
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );

    $result = $this->service->save($data);

    $this->assertNotNull($result);
    $this->assertInternalType('object', $result);
    $this->testInstanceOf('FinancaPessoal\Entity\User', $result);
}
```

Adicione o seguinte conteúdo em *src/FinancaPessoal/Entity/User.php*.

```
// src/FinancaPessoal/Entity/User.php

use FinancaPessoal\Entity\User as UserEntity;
```

```
...

/**
* @param array $data
* @return \FinancaPessoal\Entity\User $user
*/
public function save( array $data )
{
 $user = new UserEntity($data);

    $this->em->persist($user);
    $this->em->flush();

    return $user;
}
```

Ao rodarmos os testes esperamos que o mesmo passe pois a estrutura necessária está de fato pronta. Mas isso não acontece pois ainda falta criarmos a estrutura do banco de dados necessária para o teste passe.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.............E

Time: 679 ms, Memory: 6.00Mb

There was 1 error:

1) FinancaPessoal\Service\UserTest::testIfSaveIsWorkingProperly
Doctrine\DBAL\DBALException: An exception occurred while executing 'INSERT INT
O users (name, email, password, avatar) VALUES (?, ?, ?, ?)':

SQLSTATE[HY000]: General error: 1 no such table: users

/home/andre/Desktop/FLISOL/sources/vendor/doctrine/dbal/lib/Doctrine/DBAL/DBAL
Exception.php:91
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/dbal/lib/Doctrine/DBAL/Stat
ement.php:86
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/dbal/lib/Doctrine/DBAL/Conn
ection.php:647
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/orm/lib/Doctrine/ORM/Persis
ters/BasicEntityPersister.php:265
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/orm/lib/Doctrine/ORM/UnitOf
Work.php:952
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/orm/lib/Doctrine/ORM/UnitOf
Work.php:335
/home/andre/Desktop/FLISOL/sources/vendor/doctrine/orm/lib/Doctrine/ORM/Entity
Manager.php:389
```

```
/home/andre/Desktop/FLISOL/sources/src/FinancaPessoal/Service/User.php:27
/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Service/UserTest.p
hp:51

Caused by
PDOException: SQLSTATE[HY000]: General error: 1 no such table: users
...
...
```

Como já adicionamos o arquivo *tests/bootstrap.php* anteriormente falta apenas criar a estrutura das tabelas no *setUp* e destruir as mesmas no *tearDown*. Segue nova estrutura de *src/FinancaPessoal/Test/TestCase.php*.

```php
// src/FinancaPessoal/Test/TestCase.php

use Doctrine\ORM\SchemaTool;

public function setUp()
{
    $em = require dirname(dirname(dirname(__DIR__))) . '/tests/bootstrap.php';
    $this->em = $em;

    // criando todas as tabelas necessárias
    $tool = new SchemaTool($this->em);
    $classes = $this->em->getMetadataFactory()->getAllMetadata();
    $tool->createSchema($classes);

    parent::setUp();
}

public function tearDown()
{
    // destruindo todas as tabelas criadas
    $tool = new SchemaTool($this->em);
    $classes = $this->em->getMetadataFactory()->getAllMetadata();
    $tool->dropSchema($classes);

    unset($this->em);

    parent::tearDown();
}
```

Prontinho, agora se rodarmos os testes todos ele devem passar.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

..............
```

```
Time: 1.08 seconds, Memory: 7.50Mb

OK (15 tests, 28 assertions)
```

Como é perceptível, a cada vez o número de testes e asserções é acrescido, desta forma estamos cada vez mais próximos de ter uma boa parte de nossa aplicação coberta por testes e podermos evoluir a mesma de forma sadia e segura.

Testaremos agora o método *delete*.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfMethodDeleteExists()
{
    $this->assertTrue(method_exists($this->service, 'delete'));
}
```

Rodamos e o teste falha. Fazemos o mesmo passar:

```
// src/FinancaPessoal/Service/User.php

/**
* @param array $data
* @return boolean
*/
public function delete( array $data ) {}
```

Ok, o teste agora passa.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml PHPUnit 4.0.14 by Sebastian Bergma
nn.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.................

Time: 1.1 seconds, Memory: 7.50Mb

OK (16 tests, 29 assertions)
```

Em seguida precisamos verificar se o método *delete* está retornando `true` ao deletar.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfIsReturningTrueOnDelete()
{
    $this->assertTrue($this->service->delete(1));
}
```

Novamente, adicionamos a lógica necessária para que o mesmo passe em nossa classe de produção.

```
// src/FinancaPessoal/Service/User.php

/**
 *
 * @param int $id
 * @return boolean
 */
public function delete($id)
{
    $user = $this->em->getRepository('FinancaPessoal\Entity\User')->findById($
id);
    if ( $user ) {
        $this->em->remove($user);
        $this->em->flush();
        return true;
    }

    return false;
}
```

Esperamos que passe certo? Mas não é desta vez! Ainda!

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.............F

Time: 1.15 seconds, Memory: 7.75Mb

There was 1 failure:

1) FinancaPessoal\Service\UserTest::testIfIsReturningTrueOnDelete
Failed asserting that false is true.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Service/UserTest.p
hp:64

FAILURES!
Tests: 17, Assertions: 30, Failures: 1.
```

O teste falha pelo simples fato que estamos tentando remover um usuário que não existe no banco de dados. O próximo passo é criar uma maneira de ele existir no momento em que tentamos deletar. Isto pode ser feito de 4 formas.

1. Inserindo um registro instantes antes de tentar remover;
2. Inserindo um registro de usuário antes de qualquer teste (no *setUp*);
3. Utilizando Fixtures
4. Utilizando Mocks (objetos mocados, simulados)

Apenas não veremos a 3ª forma. A 4ª forma ganhará uma seção exclusiva então abordaremos a primeira e a segunda aqui.

Primeiramente altere o teste *testIfIsReturningTrueOnDelete* deixando-o como abaixo.

```
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfIsReturningTrueOnDelete()
{
    $data = array(
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );
    $this->service->save($data);

    $this->assertTrue($this->service->delete(1));
}
```

Ao rodar os testes tudo passa novamente.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.................

Time: 1.25 seconds, Memory: 7.50Mb

OK (17 tests, 30 assertions)
```

No entanto da forma feita acima é um pouco amarrada demais e tende-se a criar duplicações do código de teste tornando difícil sua manutenção. Adicionaremos agora um registro prévio no método *setUp*, desta forma ao inicializar cada um dos testes um registro já existirá no banco de dados reduzindo um pouco mais a dependência de cada um dos futuros testes.

Mova o cadastro de usuário do teste *testIfIsReturningTrueOnDelete* para *setUp*.

```
// tests/src/FinancaPessoal/Service/UserTest.php

// nova estrutura de testIfIsReturningTrueOnDelete
public function testIfIsReturningTrueOnDelete()
```

```
{
    $this->assertTrue($this->service->delete(1));
}

// nova estrutura de setUp
public function setUp()
{
    parent::setUp();
    $this->serviceName = '\FinancaPessoal\Service\User';

    $this->service = new $this->serviceName( $this->em );

    $data = array(
        'id' => 1,
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );
    $this->service->save($data);
}
```

Ok, agora rodamos os testes e tudo continua verde. A partir deste momento, qualquer outro teste futuro pode manipular o usuário que encontra-se no banco. O mesmo não se faz necessário remover no método *tearDown* por dois motivos:

1. Algum outro teste pode removê-lo, causando erro no tearDown;
2. O tearDown da classe pai destroi todas as tabelas do banco de dados;

Capítulo 4

Precisamos falar sobre Mocks.

Conceito

Mocks são basicamente objetos simulados normalmente utilizados para suprir uma necessidade de um caso de teste. Comumente é utilizado para evitar que qualquer fator externo impacte no resultado do teste.

Pense na seguinte situação:

Você está utilizando o ORM Doctrine conectado com o banco de dados Mysql em produção e com o Sqlite em ambiente de testes. Se uma das situações seguintes - um novo desenvolvedor que não possua o sqlite instalado, um novo computador que você esqueceu de instalar alguma ferramenta, ou a simples remoção da extensão do sqlite - ocorrer muito provavelmente todos os testes falharão. Outro detalhe é que juntando todos os testes que utilizam banco de dados a tendência é que os testes levem mais e mais tempo para serem executados e isto não pode

acontecer pois os testes devem ser rápidos e normalmente executados à um simples clique. Nenhum desenvolvedor gostará de rodar um teste que demore 5 minutos por exemplo.

Instalando

Assim como as demais dependências que já instalamos, utilizaremos o composer incluindo a biblioteca *Mockery* em "require-dev".

```
"require-dev" : {
    "phpunit/phpunit" : "4.*",
    "mockery/mockery": "0.9.*@dev"
}
```

> O PHPUnit possui suporte à mocks no entanto o Mockery é muito mais simples de se utlizar por isso o usaremos.

Realizamos agora o update do composer:

```
$ php composer.phar update
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing mockery/mockery (dev-master 4eff32b)
    Cloning 4eff32b268a0db52b9abe36725cb775658ad3f1f

Writing lock file
Generating autoload files
```

Pronto, já podemos utilizar o Mockery em quelquer teste que se fizer necessário.

Casos de uso

As situações mais comuns de se utilizar Mocks:

Manipulação de banco de dados

Por serem processamentos normalmente pesados mesmo que em memória como é o caso do Sqlite :memory: é uma prática muito comum utilizar mocks pelos seguintes motivos:

- Com um processamento custoso a menos o teste ocorre de forma muito mais rápida
- Se a extensão do banco de dados não estiver instalada ou qualquer outro erro que possa ocorrer os testes simplesmente quebram
- Entre outros possíveis problemas que possam ocorrer durante o desenvolvimento

Manipulação de arquivos

Um grande dificultador de testes é justamente o upload de arquivos. Com a utilização de Mocks isto pode ser sanado pois simplesmente simulamos um arquivo que será processado pelo método que realiza o upload e se comportará praticamente como se um usuário tivesse clicado no botão de upload de arquivos no browser e selecionado um arquivo de seu computador.

Disparo de E-mails

Quando estamos trabalhando em *localhost* normalmente não temos a possibilidade de disparar emails. Com mocks podemos simplesmente "Mockar" a classe de Email com todos os retornos esperados e testamos somente o que realmente nos interessa testar, afinal de contas, se você estiver utilizando o Mailer por exemplo, ele já te garante que funciona, não é preciso testar o mesmo.

Retorno de API's

Quando estamos trabalhando com API's de terceiros não podemos depender estritamente do retorno das mesmas pois o número de requisições podem ser muito grandes (dependendo da quantidade de testes) e algum problema de rede por exemplo pode simplesmente fazer tudo parar.

Utilização básica

O conceito do mock é criar um objeto simulado, fornecer um método que deseje que o mesmo execute e informar o que tal método deve retornar.

```
$mock = Mockery::mock('NomeDaClasseDesejada');

$mock->shouldReceive('nomeDoMetodo');
$mock->andReturn('valor esperado como retorno');
```

Como o Mockery possui interface fluente podemos simplificar as chamadas à nossos objetos simulados conforme abaixo

```
$mock = Mockery::mock('NomeDaClasseDesejada');

$mock->shouldReceive('nomeDoMetodo')->andReturn('valor esperado como retorno')
;
```

Quando se está trabalhando com um objeto que execute o mesmo método sob condições diferentes podemos simplesmente utilizar o os *with's* que o Mockery fornece.

```
$mock Mockery::mock('NomeDaClasseDesejada')

// Caso 1 - com parâmetro definido
```

```
$mock->shouldReceive('nomeDoMetodo')
        ->with('valor')
        ->andReturn('valor esperado');

// Caso 2 - com qualquer parâmetro
$mock->shouldReceive('NomeDaClasse')
        ->withAnyArgs()
        ->andReturn('valor esperado');

// Caso 3 - Com vários parâmetros
$mock->shouldReceive('nomeDoMetodo')
        ->withArgs(array('parametro' => 'valor'))
        ->andReturn('valor esperado');

// Caso 4 - Com nenhum parâmetro
$mock->shouldReceive('nomeDoMetodo')
        ->withNoArgs()
        ->andReturn('valor esperado);
```

Pense neste exemplo:

Atualmente estamos utilizando o Doctrine para manipular o nosso banco de dados, que tal “Mockarmos” o mesmo?

Inicialmente alteramos nossa classe *TestCase* removendo o Doctrine e utilizando o Mockery em seu lugar. Apague toda a classe já existente, copie e cole o conteúdo abaixo.

```
// src/FinancaPessoal/Test/TestCase.php

namespace FinancaPessoal\Test;

use PHPUnit_Framework_TestCase as PHPUnit;
use Mockery;


class TestCase extends PHPUnit
{

    protected $em;

    public function setUp()
    {
        $this->em = $this->getEm();

        parent::setUp();
    }

    public function tearDown()
    {
        unset($this->em);
```

```
        parent::tearDown();
    }

    public function getEm()
    {
        $em = Mockery::mock('Doctrine\ORM\EntityManager');

        $em->shouldReceive('persist')->andReturn($em);
        $em->shouldReceive('flush')->andReturn($em);
        $em->shouldReceive('remove')->andReturn($em);

        return $em;
    }
}
```

Se você rodar todos os testes existentes até o momento verá que pelo menos um deles quebrará. Nossa mas regredimos então? Não. Apenas precisamos realizar o mock de alguns métodos necessários para que os testes que quebraram passem novamente.

Resultado da execução dos testes após retiramos o Doctrine e adotarmos o Mockery:

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.............E

Time: 1.46 seconds, Memory: 6.00Mb

There was 1 error:

1) FinancaPessoal\Service\UserTest::testIfIsReturningTrueOnDelete
BadMethodCallException: Method Mockery_0_Doctrine_ORM_EntityManager::getReposi
tory() does not exist on this mock object

/home/andre/Desktop/FLISOL/sources/src/FinancaPessoal/Service/User.php:39
/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Service/UserTest.p
hp:72

FAILURES!
Tests: 17, Assertions: 29, Errors: 1.
```

Agora corrigiremos este teste que falhou para em seguida vermos um pouco mais sobre Mocks.

Estrutura antiga do método *setUp*.

```
    public function setUp()
```

```
    {
        parent::setUp();
        $this->serviceName = '\FinancaPessoal\Service\User';

        $this->service = new $this->serviceName( $this->em );

        $data = array(
            'id' => 1,
            'name' => 'teste',
            'email' => 'teste@teste.com.br',
            'password' => '1234'
        );
        $this->service->save($data);
    }
```

Nova estrutura do método *setUp*.

```
public function setUp()
{
    parent::setUp();
    $this->serviceName = '\FinancaPessoal\Service\User';

    $data = array(
        'id' => 1,
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );

    $this->em->shouldReceive('getRepository')
             ->andReturn($this->em);

    $this->em->shouldReceive('findById')
             ->andReturn(new \FinancaPessoal\Entity\User($data));

    $this->service = new $this->serviceName( $this->em );
}
```

Perceba que para o teste que quebrou passar novamente precisávamos apenas dos métodos *getRepository* e *findById* do EntityManager. Os adicionamos como *shouldReceive* e definimos o retorno esperado e tudo funciona novamente. Também removemos o cadastro inicial de um usuário que tínhamos na versão antiga do *setUp*. Agora este usuário está “mockado” por padrão fazendo com que eu não precise manipular banco de dados afinal de contas estou testando se a classe *\Service\User* está desempenhando seu papel e não se o Doctrine está funcionando.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.
```

```
Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.................

Time: 913 ms, Memory: 6.00Mb

OK (17 tests, 30 assertions)
```

> Note também que agora que além de os testes se tornarem mais próximos (se não) de unitários removemos qualquer possibilidade de erro ao rodar os testes o banco de dados interferir no resultado dos mesmos.

Realizando testes com Mocks

Agora realizaremos mais um teste utilizando Mock. O que faremos desta vez será um disparo de emails. Ao adicionarmos um usuário no sistema de finanças pessoais deve ser disparado um email de boas vindas. Obviamente que como estamos em localhost não conseguiremos enviar este email tão facilmente então "mockaremos" o mesmo.

> Utilizaremos o *mail* nativo do PHP pois é apenas um exemplo

Crie no namespace *FinancaPessoal\Email* a classe *Cadastro*.

```php
// src/FinancaPessoal/Email/Cadastro.php

namespace FinancaPessoal\Email;

class Cadastro
{

    /**
    * @param array $data
    * @return boolean
    */
    public function envia(array $data)
    {
        $subject = 'Cadastro no sistema de finanças pessoais';
        $content = 'Olá, ' . $data['name'] . '! Você foi cadastrado no sistema
';
        $content .= 'de Finanças Pessoais com o login: ' . $data['email'] . '
';
        $content .= 'e senha: ' . $data['password'];

        return true === mail($data['email'], $subject, $content);
    }
}
```

Agora nosso servico *User* precisa utilizar o email de cadastro e enviar um email para o usuário no momento do cadastro. Passaremos a classe de email como uma dependência de nosso serviço. Podemos enviar classe de email diferenciada dependendo da situação e ela não é obrigatória.

A classe de serviços do usuário sofreu algumas alterações no método *save*. Os demais métodos permanecem intactos e não foram adicionados no exemplo abaixo.

```
// src/FinancaPessoal/Service/User.php
namespace FinancaPessoal\Service;

use Doctrine\ORM\EntityManager;
use FinancaPessoal\Entity\User as UserEntity;

class User
{
    ...

    /**
     * @param array $data
     * @param \FinancaPessoal\Email\Cadastro $email
     * @return \FinancaPessoal\Entity\User $user
     */
    public function save(array $data, \FinancaPessoal\Email\Cadastro $email)
    {
        $user = new UserEntity($data);

        $this->em->persist($user);
        $this->em->flush();

        if ( $email->envia($data) ) {
            return $user;
        }
    }

    ...

}
```

Ao rodarmos os testes simplesmente o teste de cadastro de usuário quebra pois não foi possível enviar o email.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

..............sh: 1: /usr/sbin/sendmail: not found
F..
```

```
Time: 883 ms, Memory: 6.00Mb

There was 1 failure:

1) FinancaPessoal\Service\UserTest::testIfSaveIsWorkingProperly
Failed asserting that null is not null.

/home/andre/Desktop/FLISOL/sources/tests/src/FinancaPessoal/Service/UserTest.p
hp:62

FAILURES!
Tests: 17, Assertions: 28, Failures: 1.
```

Repare que há a marcação que o sendmail não foi localizado logo assim que os testes inciam no log acima. Isto é comum em ambiente de desenvolvimento local pois nem sempre temos email configurado corretamente no PHP para disparo.

Primeiramente adicione o Mockery na classe *FinancaPessoal\Service\UserTest*.

```php
// tests/src/FinancaPessoal/Service/UserTest.php

namespace FinancaPessoal\Service;

use FinancaPessoal\Test\TestCase;
use Mockery;
```

Agora criaremos o mock da classe de email de cadastro.

Estrutura antiga de *testIfSaveIsWorkingProperly*.

```php
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfSaveIsWorkingProperly()
{
    $data = array(
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );

    $result = $this->service->save($data);

    $this->assertNotNull($result);
    $this->assertInternalType('object', $result);
    $this->assertInstanceOf('FinancaPessoal\Entity\User', $result);
}
```

Nova estrutura do teste *testIfSaveIsWorkingProperly*

```
```

```php
// tests/src/FinancaPessoal/Service/UserTest.php

public function testIfSaveIsWorkingProperly()
{
    $data = array(
        'name' => 'teste',
        'email' => 'teste@teste.com.br',
        'password' => '1234'
    );

    $emailCadastro = Mockery::mock('FinancaPessoal\Email\Cadastro');
    $emailCadastro->shouldReceive('envia')->andReturn(true);

    $service = new $this->serviceName($this->em);
    $result = $service->save($data, $emailCadastro);

    $this->assertNotNull($result);
    $this->assertInternalType('object', $result);
    $this->assertInstanceOf('FinancaPessoal\Entity\User', $result);
}
```

Perceba que removemos o *$this->service* que utilizávamos antes e agora utilizamos o *$service* que foi criado para a necessidade atual, ou seja, o teste precisa ter o ambiente minimamente utilizável para ser executado de forma correta. Note antes disso que é criado o Mock da classe de email com o método e o retorno esperado do mesmo. Desta forma NÃO dependo do sendmail, o serviço é testado com sucesso caracterizando-se que o novo usuário fora gravado no banco de dados e um email de boas vindas foi enviado ao mesmo.

Agora estamos com todos os testes verdes novamente.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.................

Time: 894 ms, Memory: 6.00Mb

OK (17 tests, 30 assertions)
```

Conclusão

A intenção desta seção foi somente mostrar o que são Mocks e dar o pontapé inicial para que você os utilize em seu dia a dia. Existem muitas opções mais avançadas e situações diferenciadas para sua utilização bem como em frameworks como o Zend 2 normalmente não precisaríamos realizar a injeção de dependência da classe de email no momento do save pois o mesmo já estaria carregado no *ServiceManager* restando ao desenvolvedor apenas criar o Mock

sem se preocupar em injetá-lo onde se faz necessário o uso.

O uso de mocks pode lhe ajudar muito ao desenvolver e recomendo fortemente que busque mais fontes de estudos além deste material de apoio.

Capítulo 5

Agrupamento de testes, cobertura e recomendações.

Grupos de testes

Muitas vezes precisamos testar somente uma pequena parte de nossa aplicação pelo simples fato de termos uma sequência muito grande de testes e nem todos serem estritamente relevantes no momento. Para isto é possível agruparmos os testes. Isto é comum quando queremos testar somente nossos *Controllers*, *Services* ou *Models* por exemplo. Basta que adicionemos uma anotação em cada uma das classes de teste para que posteriormente possamos selecionar apenas um grupo a ser executado. Em resumo você anota que uma classe pertence à um grupo mas se rodar os testes sem parâmetro algum relacionado a grupos todos os testes são executados normalmente. Veja um exemplo.

```
// tests/src/FinancaPessoal/Service/UserTest.php

namespace FinancaPessoal\Service;

use FinancaPessoal\Test\TestCase;
use Mockery;

/**
 * @group Service
 */
class UserTest extends TestCase
{
    ...
}
```

A classe de teste descrita acima já pertence a ao grupo Service no entanto se rodarmos nossos testes como já vinhamos rodando, novamente todos são executados pois não passamos parâmetro de qual grupo desejamos e nem em nosso arquivo *tests/phpunit.xml* temos tal configuração.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml
```

```
.................

Time: 2.53 seconds, Memory: 6.00Mb

OK (17 tests, 30 assertions)
```

Perceba acima que temos 17 testes rodando com 30 asserções. Executaremos agora nossos testes passando como parâmetro o grupo *Service* e veremos que o resultado será diferente.

Para definir o grupo de testes que desejamos rodar basta em qualquer ponto da chamada ao PHPUnit informar o parâmetro `--group` seguido do nome do grupo desejado.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml --group Service
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

......

Time: 746 ms, Memory: 5.50Mb

OK (6 tests, 8 assertions)
```

Note que no primeiro teste deste capítulo tinhamos 17 testes com 30 asserções, agora temos 6 testes com 8 asserções. Não mexemos em nenhuma estrutura e nada que já funcionava parou de funcionar.

Existe uma segunda forma de executar os testes agrupados sem a necessidade de informar o parâmetro no momento de executarmos, basta adicionar o grupo desejado no arquivo *phpunit.xml* e toda vez que rodarmos o comando `./vendor/bin/phpunit -c tests/phpunit.xml` automaticamente rodam somente os testes definidos nos grupos. Outro detalhe é que podem ser executados testes de diversos grupos, útil para quando a aplicação está em seu início e evoluindo muito rápido no entanto existem testes que não convém serem executados no momento.

No arquivo *tests/phpunit.xml* adicione o novo conteúdo antes de fechar a tag *phpunit*.

```
<groups>
    <include>
        <group>Service</group>
    </include>
    <exclude>
        <group>Entity</group>
    </exclude>
</groups>
```

Como o xml é um formato muito bem descritivo percebe-se claramente a possibilidade de incuir e excluir grupos da bateria de testes.

Novamente rodando os testes SEM informar o parâmetro de grupos o resultado é igual o abaixo.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

......

Time: 722 ms, Memory: 5.50Mb

OK (6 tests, 8 assertions)
```

Cobertura (coverage)

Uma das ferramentas mais legais do PHPUnit é a possibilidade de coverage que nada mais é que um feedback de toda a bateria de testes que o desenvolvedor está criando. De forma clara e objetiva o desenvolvedor vê que partes do código de produção já foram testadas e quais ainda merecem sua atenção.

Assim como o agrupamento de testes podemos fazer de duas formas, diretamente através do comando de execução do PHPUnit ou através do arquivo de configurações em xml. Veremos ambos.

O coverage pode ser gerado nas seguintes saídas:

- coverage-clover - Gera uma saída em formato Clover XML
- coverage-crap4j - Gera uma saída em formato Crap4J
- coverage-html - Gera uma saída em formato HTML
- coverage-php - Gera uma classe em PHP com os reports
- coverage-text - Gera a saída em formato texto
- coverage-xml - Gera a saída em formato XML

Veremos somente o coverage em formato HTML que é de longe o mais interessante de todos mas sinta-se a vontade para testar as demais saídas.

Para que possamos gerar o coverage em formato HTML precisamos que o mesmo esteja disposto em uma pasta pois serão criados vários arquivos HTML, Javascript e CSS. Mas não se preocupe, o PHPUnit se encarrega de criar tal pasta para você, basta definí-la nas configurações de execução.

> É imprenscindível para gerar o coverage em qualquer formato que o Xdebug esteja instalado em sua máquina. Colinha: sudo apt-get install php5-xdebug

Agora finalmente geraremos nosso coverage em formato HTML nauma *tests/reports/coverage-html*.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml --coverage-html tests/reports/cove
rage-html
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

......

Time: 1.32 seconds, Memory: 7.50Mb

OK (6 tests, 8 assertions)

Generating code coverage report in HTML format ... done
```

E o resultado é o seguinte:

[imagem 1]

Perceba que nosso coverage atualmente está bem baixo, isto porque executamos somente os testes de Service com isso os demais não testaram suas respectivas classes do ambiente de produção. No arquivo *tests/phpunit.xml* remova toda a tag `<groups>` incluíndo seu conteúdo ou adicione a anotação dos grupos em todas suas classes bem como complete-as no arquivo xml. Vou imaginar que você adicionará a anotação de grupos em todas as classes de teste e configurou o xml corretamente para tal.

Agora rodamos novamente nossos testes solicitando para que seja gerado o coverage e temos o seguinte resultado:

[imagem 2]

O mais interessante do coverage em formato HTML é a possibilidade de navegação por entre o código testado. Você consegue ver por exemplo quais linhas foram testadas e quantos testes passaram por ela. Outra funcionalidade bem legal é o *Dashboard* que mostra várias estatísticas de sua aplicação. O Dashboard mostra em formato de gráficos a distribuição do coverage, a complexidade das classes testadas, as classes e métodos com cobertura insuficiente e os maiores riscos do projeto (o Crap conhecido no mundo Java). Apenas para finalizar veremos como informar detalhes do coverage em formato HTML em nosso xml de configurações. É possível definir os valores para uma baixa e alta cobertura dos testes.

Adicione o conteúdo abaixo em seu arquivo *tests/phpunit.xml* antes de `</phpunit>`.

```
<logging>
    <log type="coverage-html" target="reports/coverage-html" charset="UTF-8" y
```

```
ui="true" highlight="true" lowUpperBound="35" highLowerBound="70" />
</logging>
```

Basicamente informamos que o encoding de saída será UTF-8, que as linhas cobertas por teste serão realçadas, que um baixo valor de cobertura será abaixo de 35% e que um alto valor de cobertura será a partir de 70%.

Agora não precisamos mais passar o parâmetro `--coverage-html` quando rodarmos nossos testes.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

.................

Time: 1.85 seconds, Memory: 7.25Mb

OK (17 tests, 30 assertions)

Generating code coverage report in HTML format ... done
```

Como esperado todos os testes continuam passando e o coverage continua sendo gerado porém com muito menos parâmetros para passarmos.

Outro report bem legal é o testdox que nada mais é um *checklist* com todos os testes e os que que passaram marcados com um X. Novamente há as duas possibilidades (via parâmetro e via arquivo de configurações) no entanto veremo diretamente o arquivo de configuração em formato xml.

Dentro da tag `<logging>` adicione o log no formato testdox informando onde o mesmo será gravado. Segue nova estrutura da tag `<logging>`.

```
<logging>
    <log type="coverage-html" target="reports/coverage-html" charset="UTF-8" y
ui="true" highlight="true" lowUpperBound="35" highLowerBound="70" />
    <log type="testdox-text" target="reports/testdox.txt"/>
</logging>
```

Executamos nossos teste e temos dentro da pasta *reports* um arquivo chamado *testdox.txt* com o seguinte conteúdo:

```
FinancaPessoalEntityUser
 [x] Class exists
 [x] If method to array exists
 [x] If to array is returning an array
 [x] If to array is returning valid array
 [x] If password hash was applied
```

```
 [x] Set data andverify integrity

FinancaPessoalFilterFormataMoeda
 [x] If class exists
 [x] If method brl para float exists
 [x] Filter brl para float
 [x] If method float pata brl exists
 [x] Filter float para brl

FinancaPessoalServiceUser
 [x] Class exists
 [x] Method construct exists
 [x] If method save exists
 [x] If save is working properly
 [x] If method delete exists
 [x] If is returning true on delete
```

Basicamente o testdox lhe conta uma “história” de o que o sistema faz, desde que o desenvolvedor que criou os testes tenha um bom tato para criar nomes de métodos que descrevam o que estão fazendo. Com esse log um novo desenvolvedor pode facilmente entender tudo que o sistema faz e começar desenvolver uma nova feature facilmente.

> O testdox pode ser visualizado no momento da execução sem a necessidade de ser gravado em arquivo. Com isso no próprio terminal o programador vê o que está sendo executado.

Exemplo do testdox sendo gerado no momento dos testes no terminal.

```
$ ./vendor/bin/phpunit -c tests/phpunit.xml --testdox
PHPUnit 4.0.15 by Sebastian Bergmann.

Configuration read from /home/andre/Desktop/FLISOL/sources/tests/phpunit.xml

FinancaPessoalEntityUser
 [x] Class exists
 [x] If method to array exists
 [x] If to array is returning an array
 [x] If to array is returning valid array
 [x] If password hash was applied
 [x] Set data andverify integrity

FinancaPessoalFilterFormataMoeda
 [x] If class exists
 [x] If method brl para float exists
 [x] Filter brl para float
 [x] If method float pata brl exists
 [x] Filter float para brl

FinancaPessoalServiceUser
```

```
    [x] Class exists
    [x] Method construct exists
    [x] If method save exists
    [x] If save is working properly
    [x] If method delete exists
    [x] If is returning true on delete


  Generating code coverage report in HTML format ... done
```

Visto os reports que podemos ter com a prática do TDD percebemos claramente que temos uma ótima documentação de nossa aplicação que diferentemente de documentações tradicionais, surgem antes mesmo da implementação e se mantém praticamente 100% fiéis ao software.

Diferentes tipos de testes

O TDD é apenas uma das formas de se realizar testes de software. Além dele existem testes de integração, testes de integração, testes funcionais, testes de aceitação entre outros. A seguir abordarei de forma simplista o que vem a ser os testes listados anteriormente além do TDD.

TDD

É o menor teste possível. Ele deve ser focado em apenas uma unidade de teste, por exemplo: Verificar se o retorno de um método é um array. Independente do valor retornado, apenas devo certificar que um array está sendo retornado.

Coisas que um teste unitário NÃO deve fazer:

- Acessar recursos de rede
- Chegar/manipular o banco de dados, mesmo que em memória
- Utilizar recursos do sistema de arquivos (manipular arquivos em disco)
- Conectar-se à terceiros (APIs e afins)
- etc

Qualquer recurso que possa interferir no resultado de um teste, que seja difícil ou custoso de replicar deve ser simulado utilizando técnicas específicas para isso, como Mocks.

Testes de integração

Um pouco menos detalhado que o teste unitário, o teste de integração é o momento em que o desenvolvedor junta uma série de recursos (normalmente testados isoladamente em TDD) e garante que aquele conjunto está se comportando de maneira esperada. Outro pponto que diferencia testes de integração de testes unitários é que neste faz-se necessário conexões e manipulação de banco de dados, integra-se com recursos de terceiros, pode-se manipular arquivos. Em outras palavras cria-se uma rotina que um ser humano faria para testar o software.

A principal vantagem é que os testes de integração encontrarão erros que os testes unitários não terão conhecimento como:

- Erro de rede
- Banco de dados não disponível
- Problemas com permissões no sistema de arquivo
- Manipulação de arquivos de forma errada

As principais desvantagens são as seguintes:

- O código é maior e de mais difícil manutenção
- Erros são mais custosos de descobrir e corrigir
- A integridade dos testes é menos confiável comparado ao TDD

Testes funcionais

Diferentemente dos já citados, este tipo de teste não se preocupa com resultados intermediários ou efeitos colaterais. Basicamente sua preocupação é com relação à especificação do software, por exemplo: o método *listaItensDeUsuario* deve ser chamado passando o ID do usuário desejado e deve retornar uma lista. Basicamente é uma forma de escrever em formato de testes os casos de uso do software.

Testes de aceitação

Este tipo de teste normalmente atende todo o software. Ele testa por exemplo se ao clicar no icone da lupa o zoom é aumentado por exemplo. A vantagem é que este tipo de teste utiliza liguagem descritiva no idioma inglês. Testes de aceitação descrevem as razões porque o software deve ser criado, e não em detalhes técnicos como os tipos de testes já mencionados. No PHP utiliza-se o Behat [http://behat.org] para tais testes. A forma de testar com o Behat, bem como com outros frameworks de BDD (Behavior Driven Development) de outras linguagens é contando histórias. Exemplo: quando o usuário estiver na Home ele deve clicar no item do menu chamado contato e ser redirecionado para a página de contato. Outro exemplo. Quando o usuário chegar à tela de cadastro deve preencher todos os dados, concordar com os termos de uso, clicar no botão Concluir e ser redirecionado para a página de boas vindas. Obviamente que toda essa descrição é em inglês e há uma convenção para a mesma.

Existem ainda outros tipos de testes que não serão abordados aqui por complexidade e por sair do tema de Testes Unitários.

Como praticar TDD

Como visto ao longo deste material o TDD é um processo que exige prática e sempre fará seu código evoluir. Os próximos passos são realizar os mais diversos tipos de testes para as mais diversas situações pois o foco deste material foi apenas lhe apresentar ao TDD e ao PHPUnit a evolução na técnica continua sendo

de sua responsabilidade.

A seguir uma lista de alguns livros que podem ajudar em seus estudos:

- Test Driven Development - Kent Beck [http://goo.gl/pCndvb]
- Growing Object-Oriented Software, Guided by Tests - Steve Freeman. Nat Pryce [http://goo.gl/gHD6v9]
- Test Driven Development, Teste e Design no Mundo Real - Maurício Aniche [http://tddnomundoreal.com.br/]

Este último livro possui versões em Java e C#, em breve sairá uma versão em Ruby e se tudo ocorrer bem até o fim do ano finalmente a versão do livro para o PHP na qual estou contribuindo.

Além de livros sugiro a procurar por DOJOs que abordam por padrão a prática do TDD. Em Curitiba o Ramiro Batista da Luz @ramiroluz (que realizou a palestra “Python, por onde começar” e a oficina “Criando uma aplicação de lista de tarefas com Plone” no FLISOL) realizar mensalmente DOJOs na Aldeia Coworking.

DOJO é uma técnica de programação voltada ao raciocínio lógico e interação entre os desenvolvedores. Basicamente reunem-se desenvolvedores de várias linguagens, selecionam um problema a ser solucionado (normalmente algo que nada tenha a ver com problemas do dia a dia - tempo de cozimento de miojo contando o tempo com 3 ampulhetas com tempos diferentes por exemplo). Após selecionado o problema, os desenvolvedores seleconam uma linguagem de programação e começam a programar. Sempre de 2 em 2, um é o piloto e o outro é co-piloto, passados 7 minutos (pode variar conforme o organizador) o piloto volta para a platéia, o co-piloto vira piloto e vem um novo desenvolvedor da platéia que torna-se então o novo co-piloto. Este ciclo dura até o problema ser solucionado.

Referências

Pessoas, livros e outros que serviram de inspiração para que este material chegasse ao que chegou.

O livro do Kent Beck e também o do Maurício Aniche muito me esclaresceram no assunto TDD bem como discussões em grupos de usuários e listas de e-mails.

TDD para não técnicos - http://tdd.caelum.com.br/. Mauricio Aniche (@mauricioaniche) usa uma abordagem leve que explica de forma detalhada e com linguagem mais humana o que é o TDD e quais os benefícios o desenvolvedor e a empresa tem ao utilizar o mesmo.

Outras contribuições vieram de meu próprio blog e do Tableless nos quais tenho escrito alguns artigos e estudado muito sobre TDD. Segue link dos mesmos.

http://andrebian.com
http://tableless.com.br/author/andrecardosodev

> Written with StackEdit.