Surgiu uma nova demanda para o nosso projeto: um leilão deverá ser finalizado e, depois disso, não poderá mais ser avaliando, lançando uma exceção caso isso aconteça. Seguindo o TDD, começaremos com a implementação de um novo testLeilaoFinalizadoNaoPodeSerAvaliado(). Nele, instanciaremos o Leilao de um "Fiat 147 0KM" que receberá o lance de um Usuario "Teste" no valor de 2000.

public function testLeilaoFinalizadoNaoPodeSerAvaliado()

{

$leilao = new Leilao('Fiat 147 0KM');

$leilao->recebeLance(new Lance(new Usuario('Teste'), 2000));

}

Depois disso, chamaremos o método $leilao->finaliza() para encerrar o leilão e tentaremos avaliá-lo com o método avalia(). Nesse código estamos esperando uma exceção, portanto incluiremos os métodos expectException(), recebendo uma \DomainException, e expectExceptionMessage(), recebendo a mensagem "Leilão já finalizado".

public function testLeilaoFinalizadoNaoPodeSerAvaliado()

{

$this->expectException(\DomainException::class);

$this->expectExceptionMessage('Leilão já finalizado');

$leilao = new Leilao('Fiat 147 0KM');

$leilao->recebeLance(new Lance(new Usuario('Teste'), 2000));

$leilao->finaliza();

$this->leiloeiro->avalia($leilao);

}

Executando os testes nesse ponto teremos uma falha, afinal ainda não temos um método finaliza(). No PhpStorm, podemos selecionar o método e usar o atalho "Alt + Enter > Add Method" para o criarmos na classe Leilao. No corpo do método, simplesmente alteraremos o valor da propriedade finalizado para `true´.

public function finaliza()

{

$this->finalizado = true;

}

Como mencionado no início do treinamento, nosso foco não é no código de produção e as melhores práticas de desenvolvimento, mas sim nos testes. Poderíamos implementar um *state pattern* nesse ponto, por exemplo, mas seguiremos com a solução mais simples possível. Ainda na classe Leilao, criaremos a propriedade privada $finalizado que será inicializada no construtor com false - ou seja, um leilão sempre se inicia não-finalizado.

class Leilao

{

/\*\* @var Lance[] \*/

private $lances;

/\*\* @var string \*/

private $descricao;

/\*\* @var bool \*/

private $finalizado;

public function \_\_construct(string $descricao)

{

$this->descricao = $descricao;

$this->lances = [];

$this->finalizado = false;

}

//...

Executando novamente os testes, continuaremos com uma falha, agora na verificação de que uma exceção do tipo DomainException é lançada. No método avalia() de Avaliador, passaremos a verificar se $leilao->estaFinalizado(), um método que ainda não existe. Em caso positivo, lançaremos uma nova \DomainException com a mensagem "Leilão já finalizado".

public function avalia(Leilao $leilao): void

{

if ($leilao->estaFinalizado()) {

throw new \DomainException('Leilão já finalizado');

}

if (empty($leilao->getLances())) {

throw new \DomainException('Não é possível avaliar leilão vazio');

}

//...

Criaremos então, na classe Leilao, o método estaFinalizado(), que simplesmente retornará um booleano finalizado.

public function finaliza()

{

$this->finalizado = true;

}

public function estaFinalizado(): bool

{

return $this->finalizado;

}

Agora todos os testes passarão corretamente. Óbvio que essa implementação dá margem a diversas outras questões, por exemplo o que deve acontecer quando recebermos um lance em um leilão já finalizado, ou se é possível buscar lances em um leilão finalizado. Manteremos o nosso projeto dessa forma, com tais questões em aberto. Se você quiser, pode tentar implementar soluções para elas e, antes disso, criar testes que garantam que tais soluções funcionem.

Atualmente temos 15 métodos de teste sendo executados e mais de 30 verificações, tudo isso em um projeto fictício e bastante pequeno, garantindo que ele sempre continuará funcionando conforme evolui. Escrever testes é quase *viciante*, o que é muito bom para a saúde da aplicação. Claro, lembre-se de não gastar todo o seu tempo escrevendo testes, se esquecendo de produzir código que gera valor para o cliente de forma direta.