TDD e Testes Unitários

Test-driven Development é uma forma de fazer softwares que fazemos pequenas mudanças e testes para elas.

Com esses testes, garantimos que nosso código esta funcionando e cumpre todas as funções determinadas para ele. Além disso, esses testes funcionam como uma documentação para saber como ele funciona.

Por que usar TDD?

Existem diversas formas de fazer testes. Podemos fazer todo nosso código por exemplo, e depois escrever os testes, ou então simplesmente testar o código manualmente. TDD oferece uma metodologia que garante que todos os testes feitos sejam realmente bons, e não apenas "feitos por fazer". Os passos do TDD são:

- Escrever um **failing Test**, ou seja, garantir que o teste pode falhar, **já que testes que não falham não são muito uteis**.

- Antes de escrever um novo teste, devemos **garantir que todos os testes anteriores foram feitos com sucesso**. E esse é outro ponto importante: Todos os nossos testes são sempre repetidos, independente da **feature**, e nunca realizado apenas o teste que acabamos de fazer.

- Se fizermos todo o código para só no final colocar os testes, é provável que seja necessário refatorar ele para que consiga inserir todos os testes.

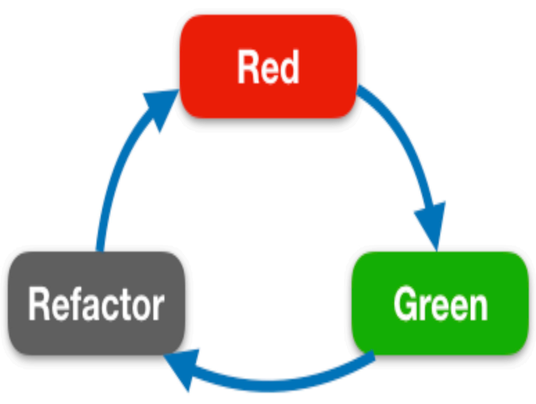
Como escrever testes?

- Escreva testes para conteúdo que você produziu, para suas classes por exemplo

- Não escreva para coisas que o Swift faz por você, e nem mesmo para coisas que são captas pelo Debugger do Xcode.

- Por fim, não escreva testes para frameworks que não são de sua autoria, isso é responsabilidade dos desenvolvedores que fizeram ela.

Ciclo do TDD



O ciclo do TDD envolve esses três passos principais:

- Vermelho: Escrever um teste capaz de falhar, antes de escrever qualquer parte do código

- Verde: Escrever o mínimo de código possível que faça passar no teste

- Refatorar: Refatorar tanto seu código quanto o teste

- Repetir: Repetir o processo até que todas features estejam implementadas.

Usando o TDD, podemos começar a montar nossos testes. Por enquanto, vamos trabalhar com eles dentro do Playgrounds.

De acordo com o ciclo do TDD, vamos começar pelo Red, escrevendo o nosso teste:

```
import XCTest

class CashRegisterTests: XCTestCase {

    func testInit_createsCashRegister() { //Nome padrão para o teste. Começamos com o test, seguido pelo metodo que vamos testar.
        XCTAssertNotNil(CashRegister()) // Garantir que nada é nil nesse test
    }
}

CashRegisterTests.defaultTestSuite.run()
```

Criamos uma classe para controlar nossos testes herdando de XCTestCase. As funções dentro dele funcionam de forma "automatica" quando adicionamos o defaultTestSuite.run(). Nesse caso de teste, vemos se não é nil o valor passado, nesse caso, CashRegister(). Porém, ainda não criamos a classe, e por isso temos o erro de compilação, então vamos criar essa classe, que faz parte do verde.

```
class CashRegister {

}
```

Agora não temos mais o erro de compilação, e podemos rodar o código. Nosso teste é rodado junto, então podemos ver no debugger que foi um sucesso.

Nota: É importante perceber que criamos um failing test no vermelho, já que erro de compilação também é um erro.

Nesse caso, não temos nada para refatorar, então vamos repetir o ciclo. Uma nova feature que vamos implementar é a de colocar um novo fundo no nosso cashRegister. Vamos criar o teste para isso.

```
    func testInitAvailableFunds_setsAvailableFunds() {

        // Dado uma certa condição
        let availableFunds = Decimal(100)

        // quando uma certa ação acontecer
        let sut = CashRegister(availableFunds: availableFunds)

        // um resultado esperado acontece
        XCTAssertEqual(sut.availableFunds, availableFunds)
    }
```

Quando fazemos um teste, podemos separar ele em três etapas principais:

- Dar uma condição
- Uma ação acontece
- Acontece um resultado

O resultado esperado é que o valor dado na classe seja igual ao declarado. Agora vamos fazer a parte verde, que consiste num init para a nossa classe **CashRegister**

```
    var availableFunds: Decimal

    init(availableFunds: Decimal = 0) {
        self.availableFunds = availableFunds
    }
```

Agora que temos esse init e testamos, podemos ver que nossos dois testes funcionam. Porém, refatorando o código, o primeiro teste se torna desnecessário, já que estamos fazendo uma inicialização no segundo teste. Então podemos remover o segundo teste.