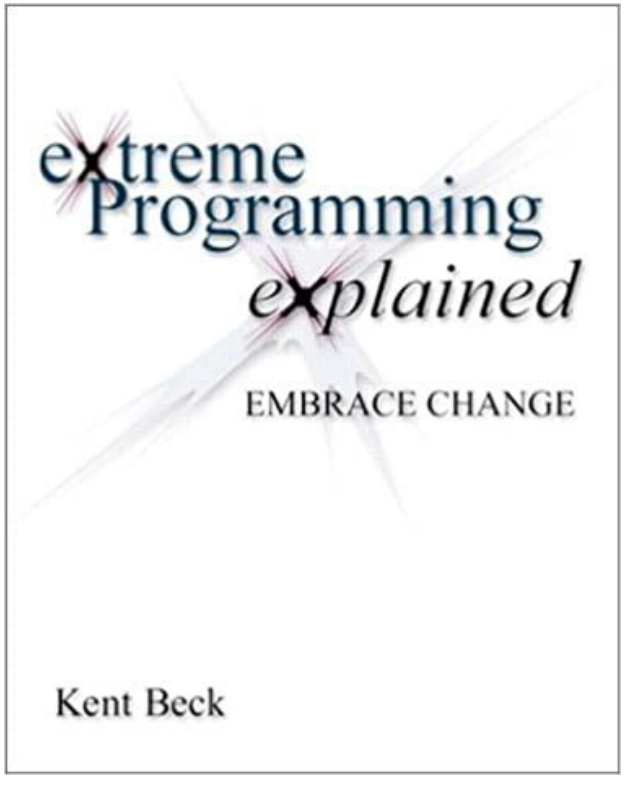
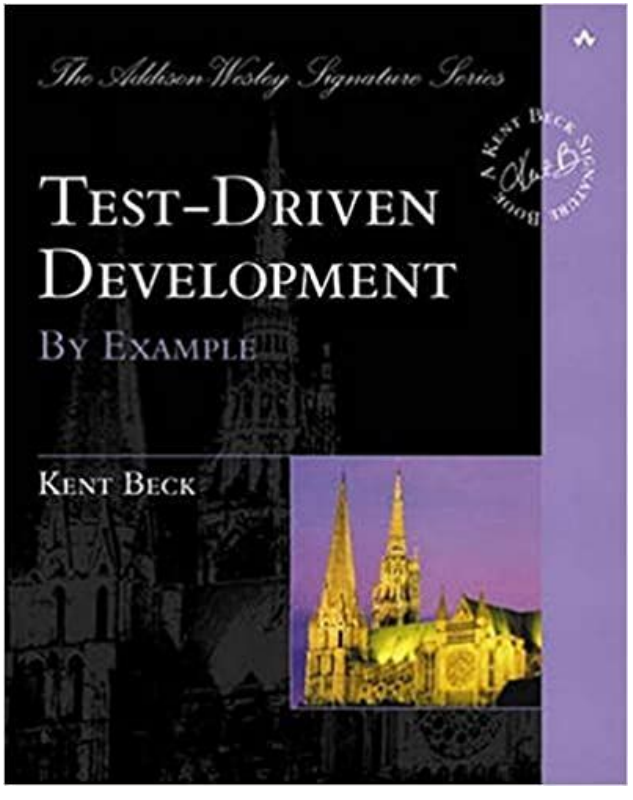
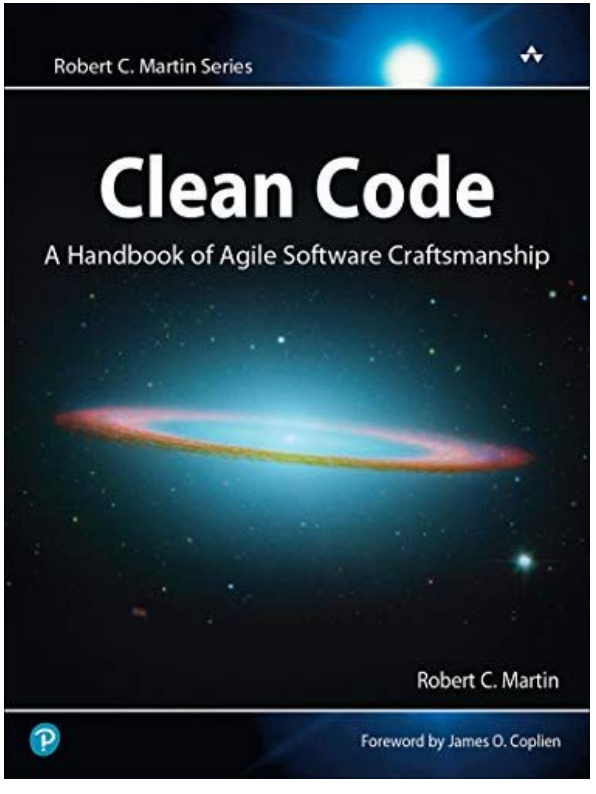
Teste de Software

Estágios do Desenvolvimento Guiado por Testes (TDD)

Lesandro Ponciano


eXtreme Programming eXplained
EMBRACE CHANGE
Kent Beck


The Addison Wesley Signature Series
A KENT BECK SIGNATURE BOOK
TEST-DRIVEN DEVELOPMENT
BY EXAMPLE
KENT BECK


Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin
Foreword by James O. Coplien

F.I.R.S.T: Características desejadas

- Testes de unidade devem ser:
  - Rápidos (***F**ast*) - testam apenas uma unidade
  - Isolados (***I**solated*) - testam cada unidade individualmente e não a integração delas
  - Repetíveis (***R**epeatable*) - resultados não mudam a cada execução dos testes
  - Auto-verificáveis (***S**elf-verifying*) – a verificação dos resultados é automática (assertTrue, assertEquals, etc)
  - Oportunos (***T**imely*) – devem ser escrito antes da implementação

Timely: Test First

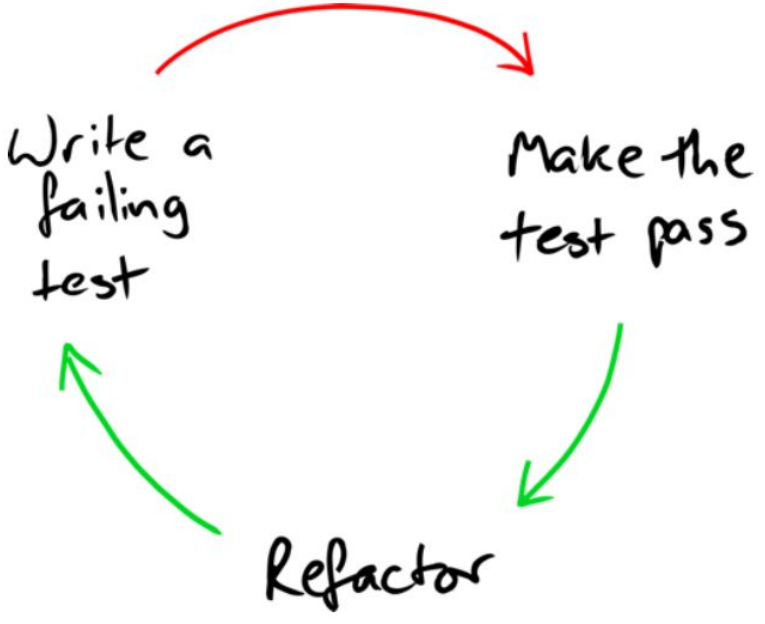
- Test-First = Escrever o teste primeiro
  - Escrever o caso de teste antes de escrever o código a que ele se refere

- Entender e projetar primeiro, implementar depois!
  - Test-first ajuda na compreensão e especificação do comportamento
  - Implicitamente, test-first ajuda a definir a interface para a qual o caso de teste está sendo criado

- É mais fácil criar o código se o teste estiver pronto

Desenvolvimento Efetivo

- Desenvolvimento Guiado por Testes
  - *Test-driven development* (TDD)
  - "*Código limpo que funciona*", por Ron Jeffries (https://ronjeffries.com/about.html)
- TDD é test-first, mas falar de test-first não é necessariamente igual a falar de TDD



Implicações Técnicas de TDD

- Desenvolvimento orgânico
  - A partir da execução do código, tomar as decisões que orientam o desenvolvimento
- O ambiente de desenvolvimento fornece respostas rápidas para pequenas mudanças
- O projeto deve ter alta coesão e baixo acoplamento para facilitar o teste
- Cada desenvolvedor deve escrever seus testes

Três Leis de TDD

1) Você não deve escrever qualquer implementação funcional antes que você tenha escrito um teste que falhe

2) Você não deve escrever mais que um teste unitário para demonstrar uma falha

3) Você não deve escrever mais do que o necessário para passar por um teste que está falhando

Por que TDD ajuda?

- Aprendizado
  - Se o defeito é descoberto alguns segundos após ser introduzido, o desenvolvedor consegue corrigi-lo rapidamente
  - Assim, pode-se aprender com o engano e passar a codificar melhor

- Redução do custo de depuração
  - Reduz o tempo entre a inserção do defeito e o momento em que ele é detectado
  - Testes expõem o local exato do defeito

Por que TDD ajuda?

- Melhoria do projeto
  - Escrever um caso de teste completo força a criação de um código desacoplado e coeso
- Documentação
  - Um caso de teste de unidade bem escrito
    - proporciona uma especificação de implementação
    - comunica a finalidade do código de forma clara
- Código nítido
  - Sua finalidade é expressa com clareza, pode ser alterado para receber novos recursos e não tem duplicação

8 Estágios de TDD

1) Escreva um teste simples
2) Compile. Não há implementação, então falhará
3) Implemente apenas o código necessário para fazer o teste compilar
4) Execute o teste e veja que ele vai falhar
5) Implemente apenas o código necessário para fazer o teste passar
6) Execute e veja o teste passar
7) Refatore o código para torná-lo mais claro
8) Repita a partir do estágio 1

Exemplo: Calculadora – Passo 1 e 2

1 - Criar um teste para a calculadora

```
class TestMethod(unittest.TestCase):

    calc=Calculadora()

    def test_case1(self):
        num1 = 1
        num2 = 1
        soma = 0
        soma = self.calc.soma(num1,num2)

        self.assertEqual(soma, 2)
```

2 - Compilar/Executar. Não executa!

Calculadora – Passo 3

3 – Código necessário para fazer compilar/executar

```
class Calculadora:

    def __init__(self):
        pass

    def soma(self, a, b):
        return 0
```

Calculadora – Passo 4

4 – Execute o teste e veja que ele vai falhar.

```
======================================================================
FAIL: test_case1 (testaCalc.TestMethod)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\lesandrop\Desktop\testaCalc.py", line 14, in test_case1
    self.assertEqual(soma, 2, "Precisa ser 2")
AssertionError: 0 != 2 : Precisa ser 2

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Calculadora – Passo 5

5 - Implemente apenas o código necessário para fazer o teste passar.

```
class Calculadora:

    def __init__(self):
        pass

    def soma(self, a, b):
        return 0
```

Antes do passo 5

```
class Calculadora:

    def __init__(self):
        pass

    def soma(self, a, b):
        return a + b
```

Após o passo 5

Calculadora – Passo 6

6 - Execute e veja o teste passar.

```
test_case1 (testaCalc.TestMethod) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Calculadora – Passo 7

7 - Refatore o código para torná-lo mais claro.

Antes do passo 7

```
class TestMethod(unittest.TestCase):

    calc=Calculadora()

    def test_case1(self):
        num1 = 1
        num2 = 1
        soma = 0
        soma = self.calc.soma(num1,num2)

        self.assertEqual(soma, 2)
```

Após o passo 7

```
class TestMethod(unittest.TestCase):

    calc = Calculadora()

    def test_caseSomaPositivo(self):
        soma = self.calc.soma(1,1)
        self.assertEqual(soma, 2, "Soma de 1 e 1 é 2")
```

Calculadora – Passo 8

Iterativamente, o código vai sendo testado e construído de modo a prover a funcionalidade

Novos casos de teste vão sendo feitos baseados nos critérios desejados, como cobertura, mutação, valores limites, etc

```
class TestMethod(unittest.TestCase):

    calc=Calculadora()

    def test_caseSomaPositivo(self):
        soma = self.calc.soma(1,1)
        self.assertEqual(soma, 2, "Soma de 1 e 1 é 2")

    def test_caseSomaNegativo(self):
        soma = self.calc.soma(-1,-1)
        self.assertEqual(soma, -2, "Soma de -1 e -1 é -2")

    def test_caseSomaZero(self):
        soma = self.calc.soma(0,0)
        self.assertEqual(soma, 0, "Soma de 0 e 0 é 0")

    def test_caseSomaPositivoNegativo(self):
        soma = self.calc.soma(2,-1)
        self.assertEqual(soma, 1, "Soma de 2 e -1 é 1")
```

Passos de Bebê no TDD - *Baby Steps*

Baby steps

- Preocupar com um caso de teste de cada vez e com um método de cada vez
  - Quando um ciclo TDD é concluído, tudo está bem compreendido
- Baby steps nos dão confiança para escrevermos determinados códigos
- Se você estiver seguro o bastante, pode acelerar o desenvolvimento
  - Se der errado, você pode voltar atrás e desacelerar

Referências

SOMMERVILLE, Ian. Engenharia de Software - 9a edição. Pearson ISBN 9788579361081.

PRESSMAN, Roger. Engenharia de software. 8. Porto Alegre ISBN 9788580555349.

BECK, Kent. TDD desenvolvimento guiado por testes. Bookman Editora, 2009.

Test First - http://www.extremeprogramming.org/rules/testfirst.html

MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

Teste de Software

Prof. Dr. Lesandro Ponciano
https://orcid.org/0000-0002-5724-0094