

TEST DRIVEN DEVELOPMENT

REAL PROGRAMMERS TEST EARLY



TESTE É COISA DE QA!
SABE DE NADA, INOCENTE

SUMÁRIO

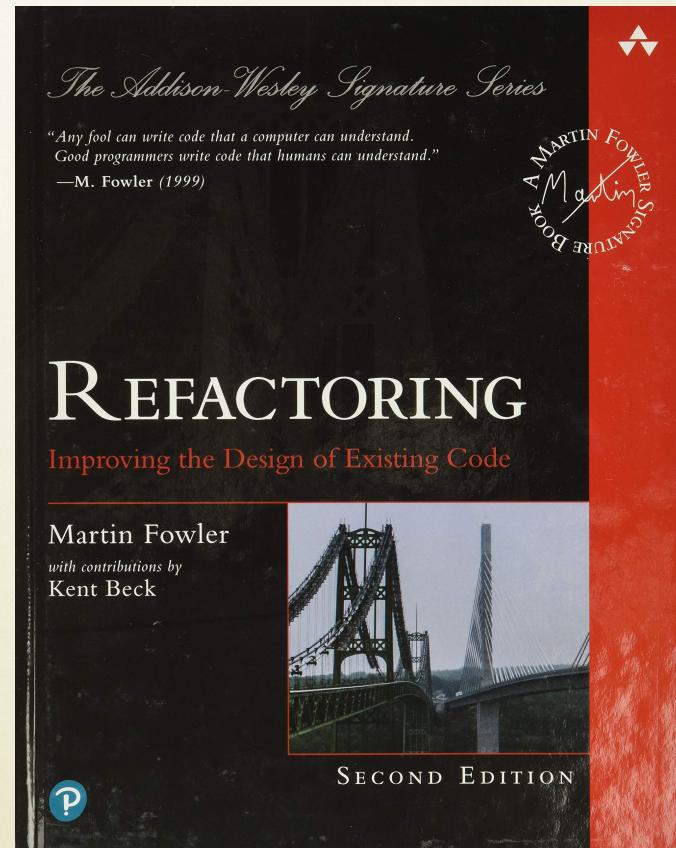
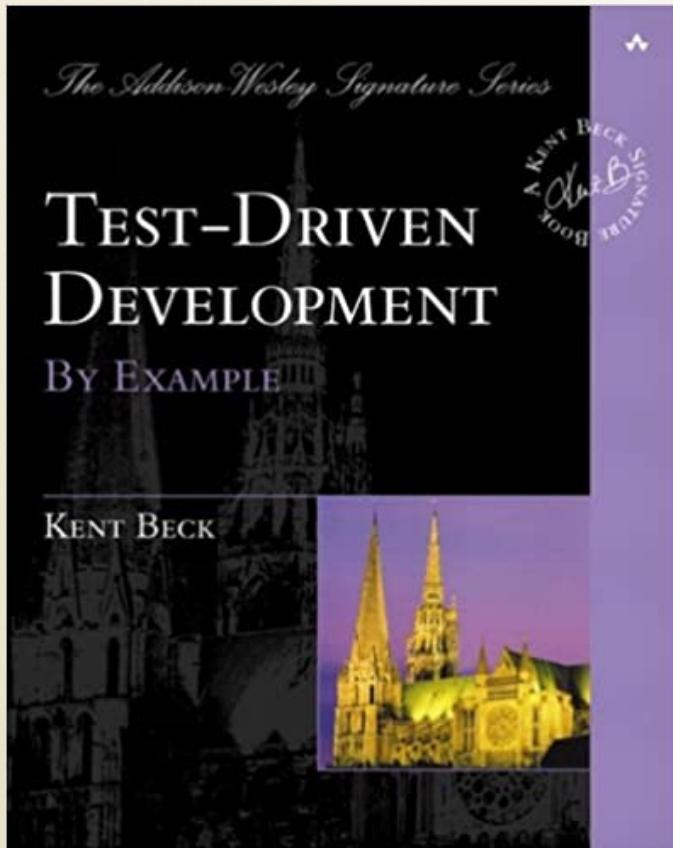
- Abordagens de Desenvolvimento
- Testes Automatizados
- Test Driven Development

AVALIAÇÃO NIVELAMENTO

- Link: <https://forms.gle/eCdBdWtfQXMbieWj8>



BIBLIOGRAFIA





ENGAGE

- **Big Idea:** Desenvolvimento de Software
- **Essential Question:** Como desenvolver software de qualidade com rapidez?
- **Challenge:** Desenvolver um pequeno protótipo em Python utilizando TDD.

INVESTIGATE

- Questões Guias: <https://bit.ly/3A1O9bX>



ABORDAGENS DE DESENVOLVIMENTO

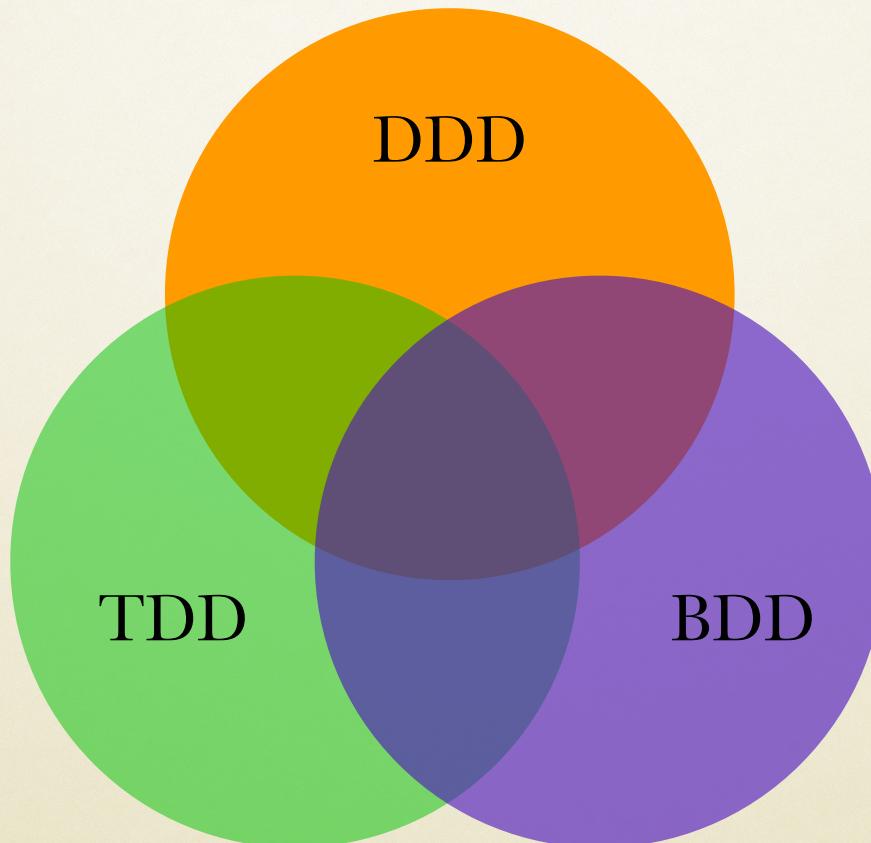
EVOLUÇÃO NO DES. DE SOFTWARE

- Características do desenvolvimento de software atual:
 - Desenvolvimento rápido para ter rápido Feedback
 - Refatoração
 - Entregas automatizadas
 - Muito código “jogado fora”

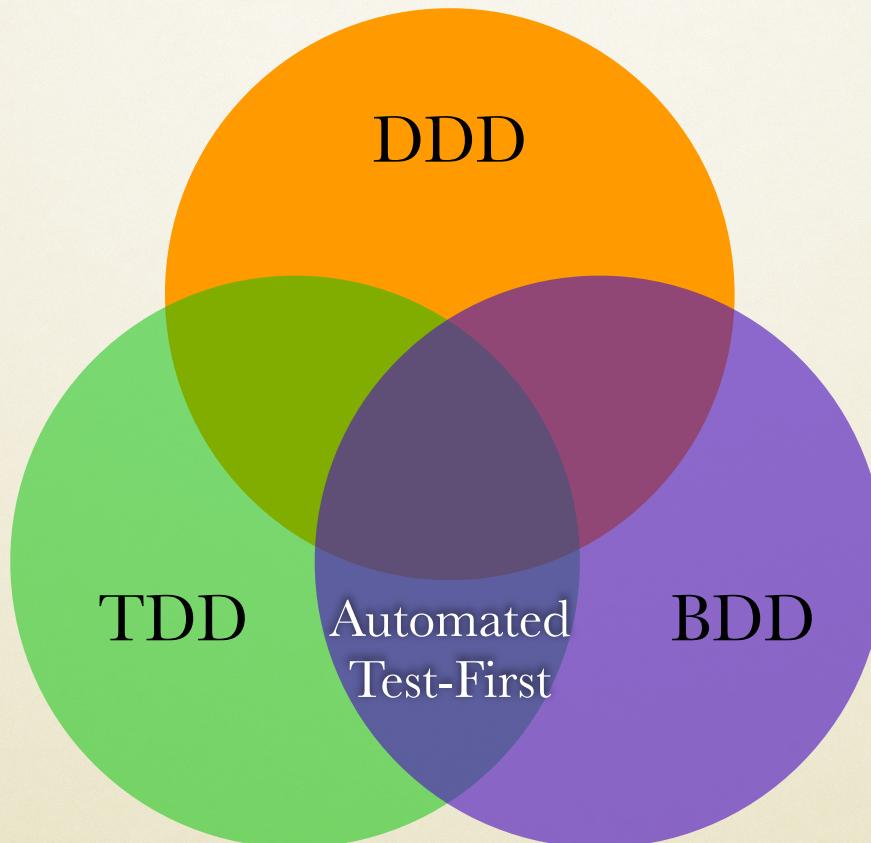
TÉCNICAS DE DESENVOLVIMENTO

- **Test Driven Development** (TDD): os testes dirigem o projeto do software e forçam a eliminação de dependências.
- **Behavior Driven Development** (BDD): construção de pedaços de funcionalidade guiada pelo comportamento esperado.
- **Domain Driven Development** (DDD): forma de projetar sistemas complexos por meio da criação de modelos e sub-modelos.

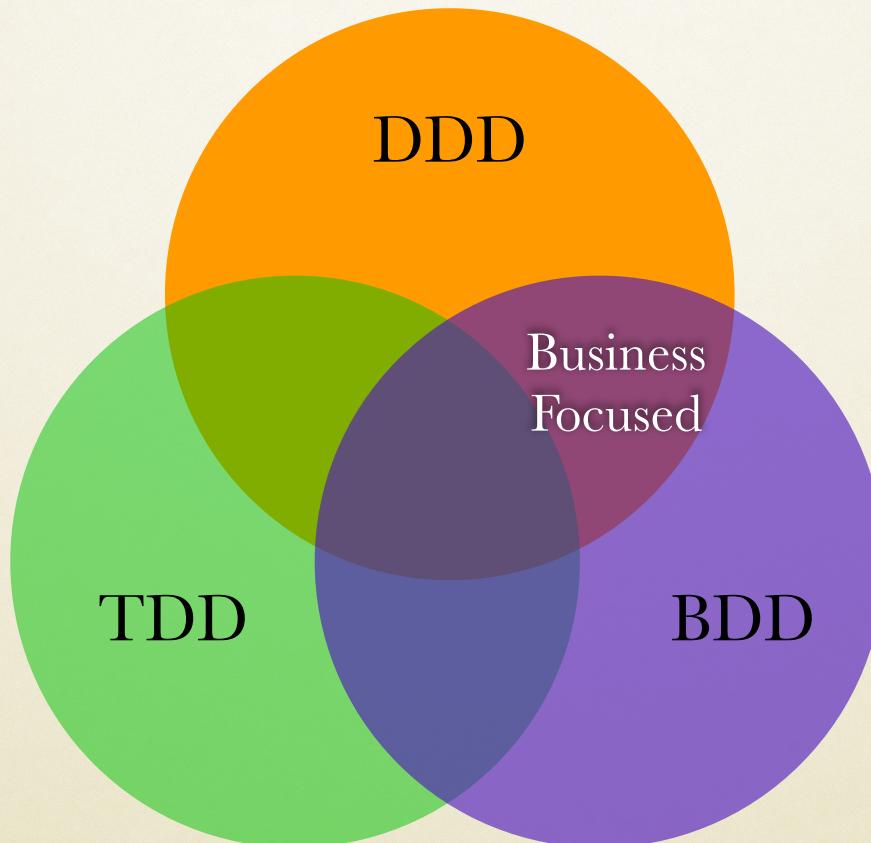
TÉCNICAS DE DESENVOLVIMENTO



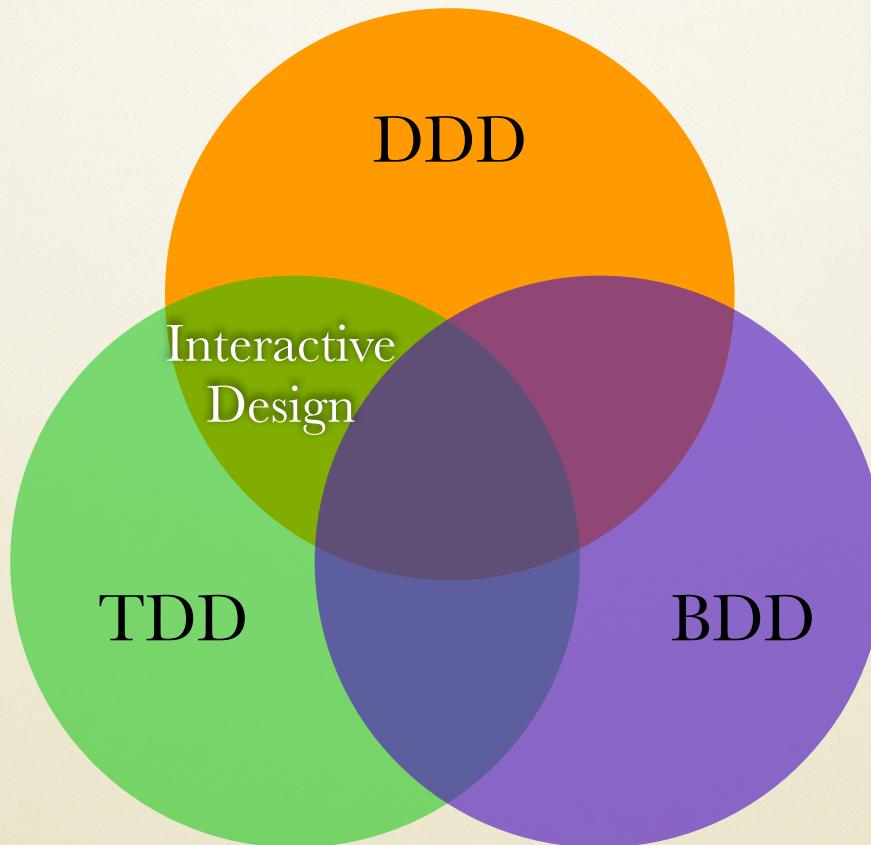
TÉCNICAS DE DESENVOLVIMENTO



TÉCNICAS DE DESENVOLVIMENTO



TÉCNICAS DE DESENVOLVIMENTO

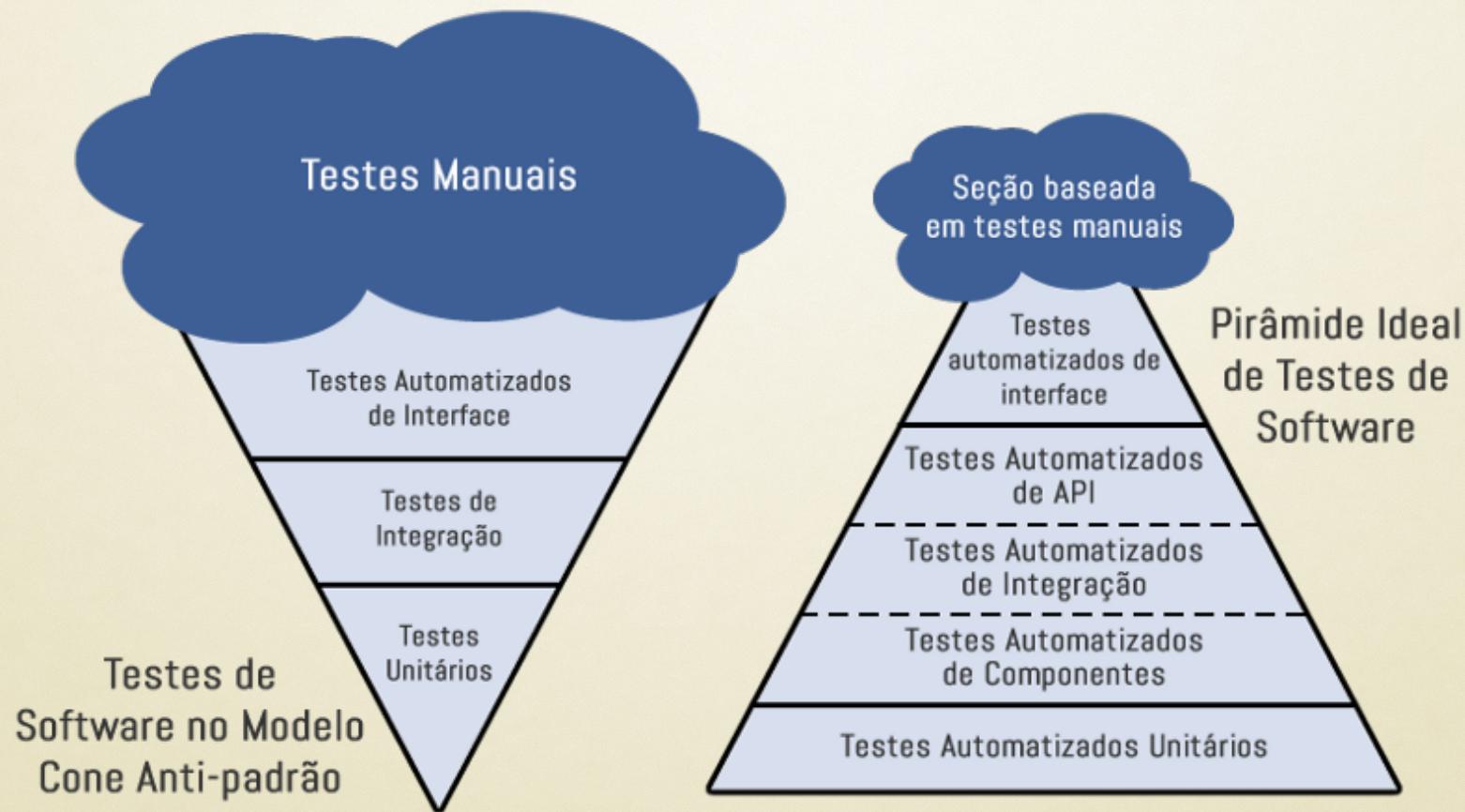


TESTES AUTOMATIZADOS

MANUAL X AUTOMATIZADO

- **Teste Manual:** uma pessoa é responsável por testar a funcionalidade do sistema como se fosse um usuário. A principal vantagem é que uma pessoa pode chegar a conclusões diferenciadas sobre os testes.
- **Teste Automatizado:** realizado por ferramentas de automação, ideal para projetos que precisam testar determinadas partes diversas vezes.

ESTRATÉGIAS DE TESTE





TESTE SEMPRE OU NÃO PROGRAME, NÃO HÁ MEIO TERMO
YODA

TESTES UNITÁRIOS

- Teste unitário é um teste feito sobre uma unidade de código.
- Unidade pode ser uma linha de código, função, propriedade, etc.

```
def calcular_salario_liquido(  
    salario_bruto,  
    ir,  
    inss,  
    sindicato):  
    return salario_bruto - (ir + inss + sindicato)
```

- Ferramenta utilizada: pytest

```
def test_calcular_salario_liquido():  
    sal_bruto = 14_000.00  
    desc_ir = 1_540.00  
    desc_inss = 1_120.00  
    desc_sind = 700.00  
    assert calcular_salario_liquido(  
        sal_bruto,  
        desc_ir,  
        desc_inss,  
        desc_sind) == 10_640.00
```

TESTES UNITÁRIOS

- Servem para:
 - Verificar se o seu trabalho está correto
 - Documentar seu código
 - TDD!
- Não devem ter dependências: usar conexões com bancos de dados, sistema de arquivos, rede, etc.

MOCKS

- Como testar código que depende de um recurso caro ou complicado?
 - Escreva uma versão “fake” do recurso cuja saída são constantes.



Brenan Keller
@brenankeller

Follow



A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM - 30 Nov 2018

TESTES DE INTEGRAÇÃO

- Testes para verificar como partes da aplicação interagem entre si.
- Não há necessidade de fazer Mocks
- Podem usar conexões com bancos de dados, sistema de arquivos, rede, etc.

TDD

PORQUE TDD?

- “A melhor defesa é o ataque”.
 - Testes escritos **antes** da produção são o **ataque**.
 - Testes escritos **depois** da produção são a **defesa**.
- TDD melhora a certeza, a coragem, a documentação, o projeto e diminui os defeitos.



3 LEIS DO TDD

1. Não programarás nenhum código de produção até ter escrito um teste que detecte uma falha
2. Não escreverás mais testes de unidade do que o suficiente para detectar uma falha (não compilar é uma falha)
3. Não implementarás mais código de produção do que o suficiente para passar no teste atual que está falhando.

SIGNIFICADO DAS 3 LEIS DO TDD

- Inicie escrevendo um pequeno teste (Lei 1).
- Em breve você vai ter que chamar uma função ou criar uma variável de uma classe que ainda não existe. O teste falha em compilar. (Lei 2)
- Você deve então escrever código de produção para fazer o teste compilar e passar.
- Como não pode escrever mais código de produção (Lei 3), deve parar e escrever mais testes.
- Os dois fluxos de código (produção e teste) crescem juntos, são complementares.

PASSOS: RED GREEN REFACTOR

1. Escrever os testes
2. Executar os testes, garantindo a falha. (RED)
3. Escrever código de produção para que os testes específicos passem. Escrever o mínimo, sem se preocupar com qualidade, refatoração virá depois.
4. Garanta que todos os testes passem. (GREEN)
5. Refatorar o código de produção. (REFACTOR)



VANTAGENS

- Obriga os desenvolvedores a pensarem nos requisitos primeiro
- Apesar de escrever mais código, no geral economiza-se tempo
- Ao escrever testes primeiro, o programador obriga-se a escrever código testável, facilitando a manutenção

DESVANTAGENS

- Em sistemas que mudam muito (startups, por exemplo), fazer todo o ciclo do TDD pode gerar perda de tempo.
- Falsa sensação de segurança: ao escrever muitos testes unitários, pode-se menosprezar outras atividades de teste, como testes de integração.
- Testes podem ser um obstáculo para mudanças: certas alterações podem quebrar muitos testes que deverão ser reescritos.
- Outra consequência do ponto acima: entender um requisito errado e escrever o teste errado. O erro só vai ser identificado mais tarde.

DESVANTAGENS

- Pontos cegos podem ocorrer quando o mesmo programador escreve os testes e o código de produção.
- Testes unitários não são críticos. Se não houver testes unitários, o cliente não vai reclamar.
- Uma solução 50% boa resolve mais problemas que uma solução 99% boa que ainda não foi publicada.



DICAS

- Cada teste deve ser independente
 - Não reutilizar dados de um teste em outro teste
- Não é necessário testar as funções da linguagem
- Compare os resultados processados com valores fixos
- Tente encontrar o equilíbrio ideal entre a velocidade de publicação de um software e a cobertura de testes. Pode ser mais eficiente lançar um produto com bugs para que se receba Feedback mais rapidamente.

**EM GERAL, NÃO SE DEVE TESTAR
DETALHES DE IMPLEMENTAÇÃO,
TESTE COMPORTAMENTOS!**

DICAS

- Não crie testes específicos para métodos ou classes.
- A origem de um teste deve ser um **requisito do sistema**.
- Crie testes para os **casos de uso** ou histórias de usuários.
- Detalhes de implementação mudam, teste apenas o contrato estável de suas APIs.
- Ao escrever um teste, primeiro imagine o melhor caso, depois trabalhe com as exceções.
- Evitar sistemas de arquivos e bancos de dados, pois dessa forma, um teste pode impactar em outro. E, caso isso seja evitado, os testes podem se tornar lentos.

REFATORAÇÃO

- Um dos pontos críticos do TDD é a refatoração.
 - É na refatoração que se escreve código limpo (Clean Code).
 - Melhora-se o código: removem-se código duplicado, code smells, aplicam-se padrões.
- Dependência é um problema chave em todos os níveis do desenvolvimento de software.
- Portanto, deve-se eliminar a dependência entre código e testes.

REFATORAÇÃO

- É o processo de mudar o software sem mudar seu comportamento externo e melhorando sua estrutura interna.
- Ao refatorar, melhora-se o código depois que ele foi escrito.

PROJETO DA DISCIPLINA

- **Big Idea:** Desenvolvimento de Software
- **Engage:** Como desenvolver software de qualidade com rapidez?
- **Challenge:** Desenvolver um pequeno protótipo em Python utilizando TDD.
- Entrega: Arquivos .py de produção e arquivos .py dos testes
- Data: 23/08