DESENVOLVIMENTO GUIADO POR TESTES (TDD – TEST DRIVEN DEVELOPMENT)

Prof. Msc. Rober Marcone Rosi

SUMÁRIO

- 5.1 - Definição
- 5.2 - Benefícios e Limitações
- 5.3 - Ciclo de Desenvolvimento
- 5.4 - Padrões para Desenvolvimento Guiado por Testes

INTRODUÇÃO

- Com o surgimento das metodologias ágeis na década de 2000, houve a necessidade do aperfeiçoamento das técnicas de testar software.
- Já não fazia mais sentido os testes no final, pois as entregas eram constantes.
- Houve a adoção da abordagem de iterações para os testes também, não esperando para testar quando o produto ficasse completamente pronto.

DEFINIÇÃO

- A ideia por traz do desenvolvimento guiado por teste é que primeiro devemos escrever os testes para posteriormente escrever o código.
- Desenvolvimento guiado por testes é uma técnica de desenvolvimento de software que baseia em um ciclo curto de repetições: primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade.
- Então, é produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis.
- O Test Driven Development (TDD) é parte do processo de desenvolvimento ágil, utilizado em metodologias como o XP (Programação Extrema) e sendo uma das técnicas que auxiliam na melhoria de qualidade do processo de desenvolvimento.
- O TDD ou desenvolvimento guiado por teste torna mais eficiente o processo.

DEFINIÇÃO

- Esse desenvolvimento é altamente iterativo e baseado em ciclos de desenvolvimento de casos de teste automatizados, em seguida, na criação e integração de pequenos trechos de código, executando os testes de componentes, corrigindo quaisquer problemas e regerando o código.
- Esse processo continua até que o componente tenha sido completamente construído e todos os testes de componentes tenham passado.
- O desenvolvimento guiado por teste é um exemplo de uma abordagem de teste inicial.

DEFINIÇÃO

- Desenvolvimento Guiado por Testes é um processo que modifica o paradigma do desenvolvimento de softwares tradicional.
- Em vez de desenvolver, primeiramente, seu código e ajustá-lo de maneira retroativa para validá-lo, o TDD determina que os testes sejam escritos antes e que as adaptações sejam, só depois, aplicadas ao código até que o projeto atenda aos requisitos do teste já definido.

DEFINIÇÃO

- Regras fundamentais do TDD:
  - Escreva o teste da implementação ANTES de escrevê-la.
  - Escreva somente código suficiente para o teste passar e nada além disto.
  - Escreva testes pequenos: teste a maior quantidade possível de código de cada vez.
  - Escreva testes muito rápidos: não devem demorar mais do que alguns segundos para serem executados.

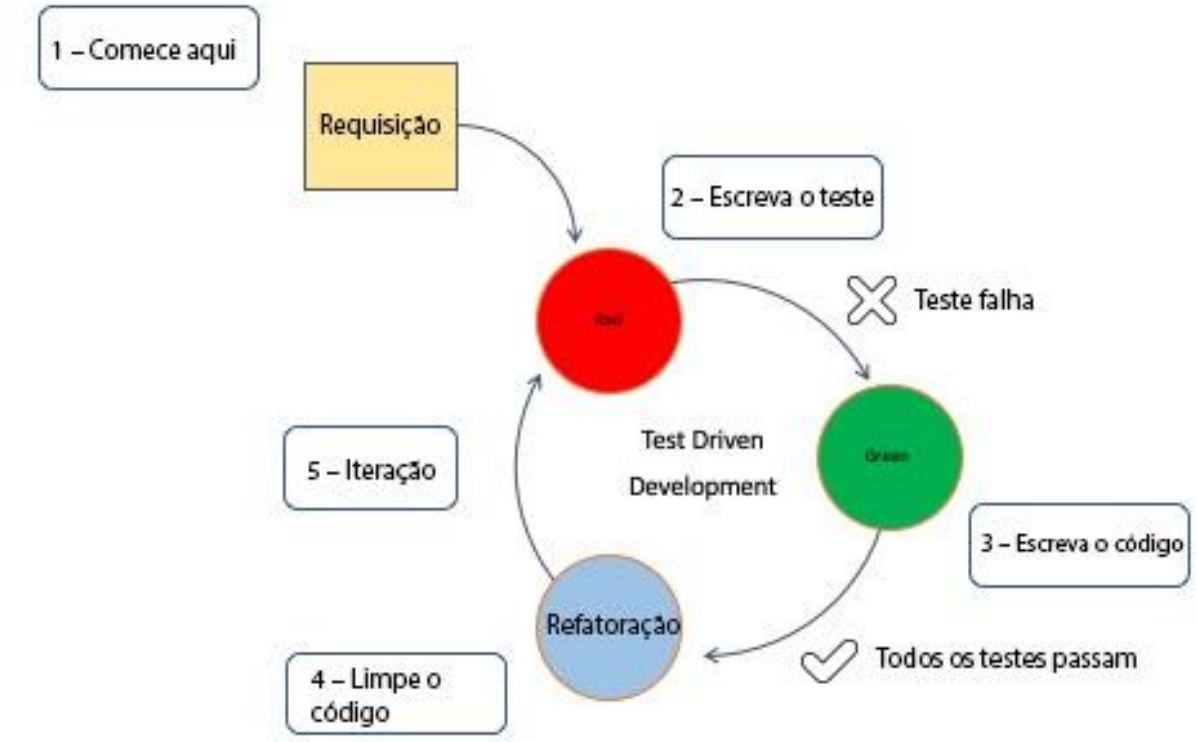
MOTIVAÇÃO

- Design pouco testável.
- Baixa cobertura de testes unitários.
- Necessidade de levantar todo o ambiente e testar.
- Necessidade de manter compatibilidade retroativa.
- Insegurança ao modificar base do código.

ETAPAS DA PROGRAMAÇÃO COM TDD

1. Criar um teste
2. Executar todos os testes da aplicação para ver o novo teste falhar.
3. Escrever a implementação testada.
4. Executar os testes para ver se todos passarão.
5. Refatoração.
6. Executar os testes novamente para garantir que eles continuam passando.

ETAPAS DA PROGRAMAÇÃO COM TDD



TIPOS DE TESTE

- Testes unitários
- Testes de integração
- Testes de aceitação

TESTES UNITÁRIOS

- Testam apenas um componente do sistema.
- O Teste Unitário valida apenas um pedaço do sistema, ou seja, uma unidade, sendo utilizado para testar as funcionalidades, exibindo informações sobre seu funcionamento.
- Todos os outros componentes são simulados (mock objects)
- Ex: ferramentas: Junit, JMock

TESTES UNITÁRIOS

- Testes de unidades são assim chamados por cada teste exercitar uma unidade de código.
- Se um módulo tem centenas de testes de unidade ou somente cinco é irrelevante.
- Um conjunto de testes em TDD nunca cruza os limites de um programa, nem deixa conexões de rede perdidas.
  - Ao fazer essas ações, o mesmo introduz intervalos de tempo que podem tornar testes lentos ao serem executados, desencorajando desenvolvedores de executar toda a suite de testes.
  - Introduzir dependências de módulos externos ou data transforma teste de unidade em testes de integração.
  - Se um módulo se comporta mal em uma cadeia de módulos interrelacionados, não fica imediatamente claro onde olhar a causa da falha.

TESTES DE INTEGRAÇÃO

- Testam a integração entre os componentes
- Envolvem dois ou mais componentes (classes + SGBD, classes + SGDB + webservices, várias camadas da aplicação)
- Ex: Ferramentas: Junit, DBUnit, HSLQDB

TESTES DE INTEGRAÇÃO

- Quando o código em desenvolvimento depende confiavelmente de uma base de dados, um serviço web ou qualquer outro processo externo ou serviço, dois passos são necessários:
  - Uma interface deve ser definida de forma a descrever que acessos irão ser disponíveis.
  - O princípio de inversão de dependência (baixo acoplamento de alto para baixo nível) fornece benefícios nessa situação em TDD.
  - A interface deve ser implementada de duas maneiras, uma que realmente acessa o processo externo, e outra que é um fake ou mock.
  - Objetos fake precisam fazer um pouco mais do que adicionar mensagens "Objeto Pessoa salvo" criando registro de rastreio, identificando que asserções podem executadas para verificar o comportamento correto.
  - Objetos mock diferem pelo fato que eles mesmos contém asserções que podem fazer com que o teste falhe. Exemplo:
    - Se o nome da pessoa e outro dado não é como esperado, métodos de objetos fake e mock que retornem dados, aparentemente de uma armazenamento de dados ou de usuário, pode ajudar ao processo de teste sempre retornar o mesmo, dados que testes confiáveis.
    - Implementações de fakes e mocks são exemplos de injeção de dependência.
    - A proposta da injeção de dependência (baixo acoplamento) é que a base de dados ou qualquer ou código de acesso externo nunca é testado pelo processo TDD.
    - Para desviar de erros que possam aparecer em função disso, outros testes com a real implementação das interfaces discutidas acima são necessários.

TESTE DE ACEITAÇÃO

- Testam uma história, funcionalidade ou caso de uso.
- Envolvem vários componentes do sistema.
- Ex: Ferramentas: Junit, Selenium, Fit / FitNesse

BENEFÍCIOS

- O desenvolvimento guiado por teste dá uma visão mais ampla do que deve ser feito ao desenvolvedor, pois antes de criar a funcionalidade, deve-se criar um teste da funcionalidade.
- Quando é criado o teste, normalmente quer dizer que foi entendido o que é para se desenvolver.
- Os passos apresentados indicam que primeiro é necessário criar o teste, este teste é chamado de teste falho, pois a funcionalidade ainda não existe, desta forma, o teste deve retornar um erro.
- Posteriormente será desenvolvido o código para fazer com que o teste seja bem sucedido, já que o desenvolvedor sabe quais funcionalidades deve implementar, fica mais prático o seu desenvolvimento.
- Por último refatore, ou seja, melhore a codificação.
- O Desenvolvimento guiado por teste é um conjunto de técnicas que culminam em um teste de ponta a ponta.

LIMITAÇÕES

- Desenvolvimento dirigido com testes é difícil de usar em situações onde testes totalmente funcionais são requeridos para determinar o sucesso ou falha. Exemplos disso são interfaces de usuários, programas que trabalham com base de dados, e muitos outros que dependem de configurações específicas de rede.
- Suporte gerencial é essencial.
- Se toda a organização não acreditar que TDD é para melhorar o produto, o gerenciamento irá considerar que o tempo gasto escrevendo teste é desperdício.
- Os próprios testes se tornam parte da manutenção do projeto.
- Testes mal escritos, por exemplo, que incluem strings de erro embutidas ou aqueles que são susceptíveis a falha, são caros de manter.
- Há um risco em testes que geram falsas falhas de tenderem a serem ignorados. Assim quando uma falha real ocorre, ela pode não ser detectada.

LIMITAÇÕES

- O nível de cobertura e detalhamento de teste alcançado durante repetitivos ciclos de TDD não pode ser facilmente recriado em uma data tardia.
- Lacunas inesperadas na cobertura de teste podem existir ou ocorrer por uma série de razões.
- Talvez um ou mais desenvolvedores em uma equipe não foram submetidos ao uso de TDD e não escrevem testes apropriadamente, talvez muitos conjuntos de testes foram invalidados, excluídos ou desabilitados acidentalmente ou com o intuito de melhorá-los posteriormente. Se isso acontece, a certeza é de que um enorme conjunto de testes TDD serão corrigidos tardiamente e refatorações serão mal acopladas.
- Alterações podem ser feitas não resultando em falhas, entretanto, na verdade, bugs estão sendo introduzidos imperceptivelmente, permanecendo indetectáveis.

LIMITAÇÕES

- Testes de unidade criados em um ambiente de desenvolvimento dirigido por testes são tipicamente criados pelo desenvolvedor que irá então escrever o código que está sendo testado.
- Os testes podem consequentemente compartilhar os 'pontos cegos' no código: Se por exemplo, um desenvolvedor não realizar determinadas entradas de parâmetros que precisam ser checadas, muito provavelmente nem o teste nem o código irá verificar essas entradas.
- Logo, se o desenvolvedor interpreta mal a especificação dos requisitos para o módulo que está sendo desenvolvido, tanto os testes como o código estarão errados.
- O alto número de testes de unidades pode trazer um falso senso de segurança, resultando em menor nível de atividades de garantia de qualidade, como testes de integração e aceitação.

PADRÕES PARA DESENVOLVIMENTO GUIADO POR TESTES

- **Teste isolado (Isolated Test)**
  - Primeiro, faça testes tão rápidos de executar que possa rodá-los sozinho e rodá-los frequentemente.
  - Testes deveriam ser capazes de ignorar um ao outro completamente.
  - Uma implicação conveniente de testes isolados é que os testes são independentes de ordem.
  - Se quero pegar um subconjunto de testes e rodá-los, então eu posso fazê-lo sem me preocupar com que o teste vá falhar agora porque um teste que é pré-requisito sumiu.
  - Isolamento de testes o encoraja a compor soluções de muitos objetos altamente coesos e fracamente acoplados.

Padrões para Desenvolvimento Guiado por Testes

- **Teste primeiro (Test First)**
  - Quando você deveria escrever seus testes? Antes de escrever o código que vai ser testado.
- **Defina uma asserção primeiro (Assert First)**
  - Por onde você deve começar a construir um sistema? Com histórias de que quer ser capaz de contar sobre o sistema terminado.
  - Por onde você deve começar a escrever um pedaço de funcionalidade? Com testes que quer que passem com o código terminado.
  - Por onde você deve começar a escrever um teste? Com as asserções que passarão (serão bem-sucedidas) quando estiver feito.

PADRÕES PARA DESENVOLVIMENTO GUIADO POR TESTES

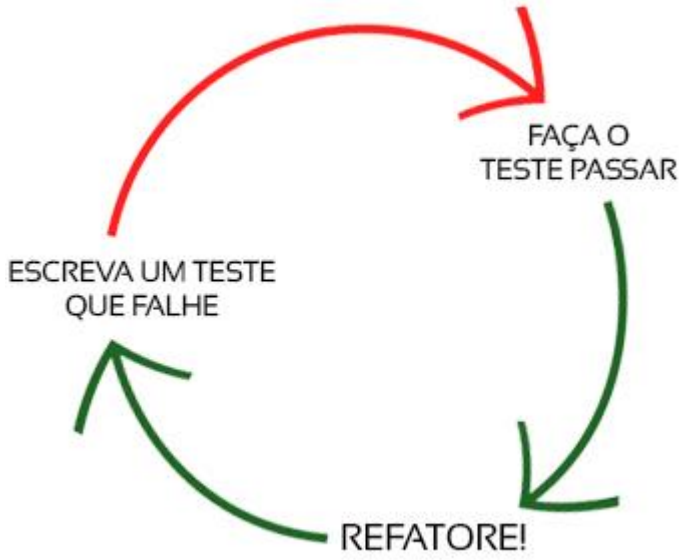
- **Dados de teste (Test Data)**
  - Use dados que façam os testes fáceis de ler e seguir.
  - Dados de Teste não são uma licença para uma parada repentina na confiança plena.
  - Se seu sistema tem que manipular múltiplas entradas, então seus testes deveriam refletir múltiplas entradas.
  - Contudo, não tenha uma lista de dez itens como entradas se uma lista de três itens guiarão você às mesmas decisões de projeto e implementação.

PADRÕES PARA DESENVOLVIMENTO GUIADO POR TESTES

- **Dados evidentes (Evident Data)**
  - Inclua resultados esperados e reais no próprio teste e tente fazer seu relacionamento evidente.
  - Um efeito colateral benéfico de Dados Evidentes é que ele torna a programação mais fácil.
  - Uma vez que tenhamos escrito a expressão na asserção, sabemos o que precisamos programar.

PADRÕES DE BARRA VERMELHA

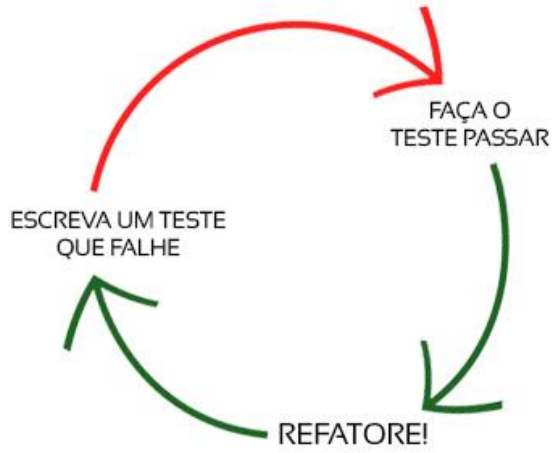
- **Barra vermelha:** Testes que falharam.
- **Teste de um só passo (One Step Test)**
  - Qual é o próximo teste que você deveria pegar da lista? Pegue um teste que ensinará a você algo e que você confia que pode implementar.
- **Teste Inicial (Starter Test)**
  - Com qual teste você deveria começar? Comece testando uma variante de uma operação que não faz nada.
  - Pegue um Teste Inicial que o ensinará alguma coisa, mas que você está certo que pode fazer funcionar rapidamente.
  - Se está implementando uma aplicação pela enésima vez, então pegue um teste que requer uma operação ou duas.
  - Você estará, justificadamente, confiante de que pode fazê-lo funcionar.



PADRÕES DE BARRA VERMELHA

- **Teste de explicação (Explanation Test)**

  - Como você difunde o uso de teste automatizado? Peça e dê explicações em termos de testes.
  - Você pode fazer isso em mais altos níveis de abstração. Se alguém está explicando um diagrama de sequência para você, então pode pedir permissão para convertê-lo em uma notação mais familiar.
  - Então você digita um caso de teste que contenha todos os objetos e variáveis externamente visíveis no diagrama

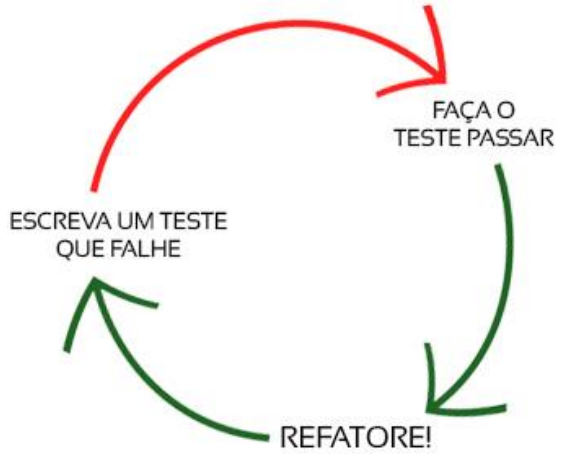


PADRÕES DE BARRA VERMELHA

- **Teste de aprendizado (Learning Test)**
  - Quando você escreve testes para software produzido externamente? Antes da primeira vez que você vai usar uma nova versão do pacote.
  - Uma alternativa é perceber que estamos prestes a usar um novo método de uma classe.
  - Em vez de apenas usá-lo, escrevemos um pequeno teste que verifica que a API funciona como esperado.
- **Outro teste (Another Test)**
  - Como você evita que uma discussão técnica desvie do assunto? Quando uma ideia tangencial emerge, adicione um teste à lista e volte ao assunto.
- **Teste de regressão (Regression Test)**
  - Qual é a primeira coisa que você faz quando um defeito é informado? Escreva o menor teste possível que falhe e que, uma vez rodado, será reparado.

PADRÕES DE BARRA VERMELHA

- **Pausa (Break)**
  - O que você faz quando se sente cansado ou travado? Faça uma pausa.
- **Faça de novo (Do Over)**
  - O que você faz quando se sente perdido? Jogue o código fora e comece de novo.
- **Mesa barata, cadeira legal (Cheap Desk, Nice Chair)**
  - Qual configuração física você deveria usar para TDD? Tenha uma cadeira realmente legal, poupando no resto dos móveis se necessário.



PADRÕES DE TESTES

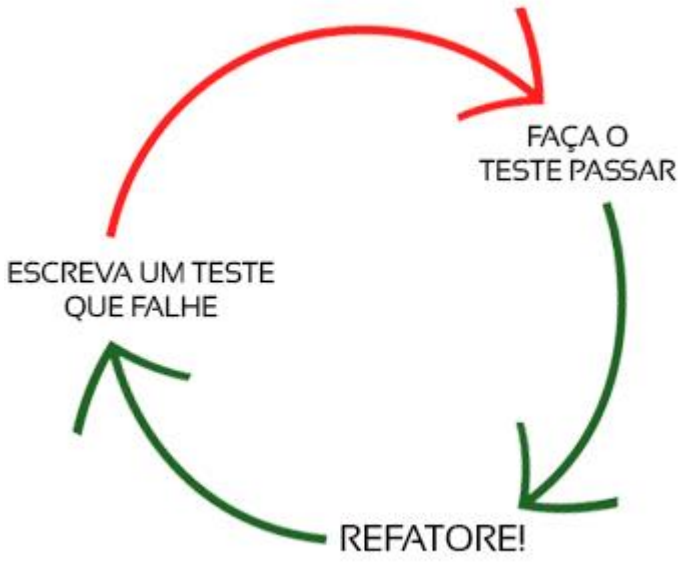
- **Teste filho (Child Test)**
  - Como você executa um caso de teste que se mostrou muito grande? Escreva um caso de teste menor que represente a parte que não funciona do caso de teste maior.
  - Consiga rodar o caso de teste menor.
  - Reintroduza o caso de teste maior.
- **Objeto simulado (Mock Object)**
  - Como você testa um objeto que se baseia em um recurso caro ou complicado? Crie uma versão faz de conta do recurso que responde com constantes.
- **Autodesvio (Self Shunt)**
  - Como você testa se um objeto se comunica corretamente com outro? Tenha o objeto que está sob teste se comunicando com o caso de teste, em vez do objeto que ele espera.

PADRÕES DE TESTES

- **String de registro (Log String)**
  - Como você testa se a sequência em que as mensagens são chamadas está correta? Mantenha um registro (log) em uma string e, quando uma mensagem for chamada, acrescente-a à string.
- **Modelo de teste de acidentes (Crash Test Dummy)**
  - Como você testa código de erro que provavelmente será pouco invocado? Invoque- o de qualquer forma com um objeto especial que lança uma exceção em vez de fazer trabalho real.
  - Código que não é testado não funciona.
- **Teste quebrado (Broken Test)**
  - Como você deixa uma sessão de programação quando está programando sozinho? Deixe o último teste quebrado.

PADRÕES DE TESTES

- **Check-in limpo (Clean Check-in)**
  - Como você deixa uma sessão de programação quando está programando em um time? Deixe todos os testes rodando.

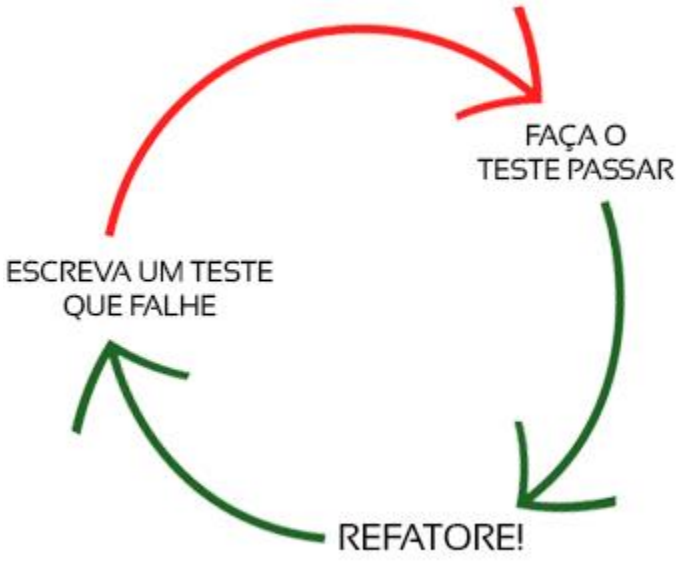


PADRÕES DE BARRA VERDE

- Depois de ter um teste não funcionando, você precisa arrumá-lo.
- Se você trata uma barra vermelha como uma condição a ser corrigida o mais rápido possível, então descobrirá que pode ter o verde rapidamente.
- Use esses padrões para fazer o código funcionar.
- **Fazer de conta (até fazê-lo) – Fake It (Til You Make It)**
  - Qual é sua primeira implementação uma vez que tem um teste que não funciona?
  - Retorne uma constante. Depois de ter o teste rodando, gradualmente transforme a constante em uma expressão usando variáveis.
- **Triangular (Triangulate)**
  - Como você conduz abstração com testes de forma mais conservadora? Abstraia apenas quando tiver dois ou mais exemplos.
- **Implementação óbvia (Obvious Implementation)**
  - Como você implementa operações simples? Apenas implemente-as.

PADRÕES DE BARRA VERDE

- **Um para muitos (One to Many)**
  - Como você implementa uma operação que funciona com coleções de objetos? Implemente-a sem as coleções primeiro, então a faça funcionar com coleções.



REFERÊNCIAS

- Baseado no material da Profa. Denise Togneri e Ralf Luís de Moura