

# Testes de Software – Prof. Eiji Adachi

---

## Trabalho da 3ª Unidade – Testes de Muta  o e Testes com Dubl  s (Fakes e Mocks)

### 1. Contexto

Este trabalho   uma continua  o direta do enunciado da 2  unidade, no qual voc  testou o m todo `calcularCustoTotal()` de uma aplica  o de e-commerce.

Nesta etapa, voc  dever  testar uma vers o simplificada do c lculo de custo total e, adicionalmente, criar testes para o m todo `finalizarCompra()`, utilizando fakes e mocks para simular depend ncias.

O trabalho pode ser realizado individualmente ou em dupla.

---

## 2. M todo Simplificado `calcularCustoTotal()`

---

Para esta unidade, voc  dever  implementar novamente o m todo de calcular custo total, mas desta vez uma vers o reduzida que segue apenas as regras abaixo.

### 2.1 Regra de Desconto

- Apenas uma regra: **Desconto por valor total do carrinho**
- Se valor total  $\geq$  R\$ 1000,00, ent o aplique 20% de desconto.
  - Se valor total  $\geq$  R\$ 500,00 &&  $<$  R\$ 1000,00, ent o aplique 10% de desconto.
  - Outros valores: sem desconto.

### 2.2 Regra de Frete

Com o peso total calculado com base apenas no peso f sico, aplica-se a seguinte tabela:

Faixa	Peso total	Valor do frete
A	0–5 kg	frete isento (R\$ 0)
B	> 5 kg e $\leq$ 10 kg	R\$ 2,00 por kg
C	> 10 kg e $\leq$ 50 kg	R\$ 4,00 por kg
D	> 50 kg	R\$ 7,00 por kg

#### Observa  es importantes:

- Produtos fr geis:** Para cada item marcado como “fr gil”, s o cobrados **R\$ 5,00 adicionais por unidade**.
- N o existe adicional por regi o nem desconto por perfil de fidelidade (cliente Ouro, Prata ou Bronze).

## 2.3 Ordem de Cálculo

1. Subtotal = soma do preço unitário multiplicado pela quantidade de cada item.
2. Aplicar o desconto conforme a regra acima.
3. Calcular o frete conforme a regra acima.
4. Total = subtotalComDesconto + frete.
5. O valor final deve ser arredondado para duas casas decimais.

---

## 3. Testes Obrigatórios para `calcularCustoTotal()`

---

### 3.1 Cobertura Estrutural

Os testes devem atingir obrigatoriamente:

- 100% de cobertura de arestas (*branch coverage*).

O relatório do JaCoCo deve demonstrar claramente que todas as arestas do método foram cobertas.

### 3.2 Mutação

É obrigatório:

- Utilizar PITEST para análise de mutação.
- Atingir 100% de mutantes mortos no método.
- Documentar no README:
  - Linha de comando usada.
  - Como verificar que não restaram mutantes sobreviventes.
  - Estratégias usadas para matar mutantes sobreviventes.

---

## 4. Testes do Método `finalizarCompra()`

---

Além dos testes estruturais do cálculo, você deverá criar testes para o método `finalizarCompra()`, garantindo a simulação de todo o comportamento usando dublês adequados.

### 4.1 Objetivo do Teste

Criar testes automatizados cobrindo o fluxo completo:

1. Verificação de estoque via serviço externo.
2. Cálculo do custo total.
3. Autorização de pagamento.
4. Atualização do estoque e finalização da compra.

Os testes devem atingir 100% de cobertura de decisão do método `finalizarCompra()`. Devem verificar também:

- Se os métodos esperados foram invocados.

- Se o resultado retornado corresponde ao comportamento especificado.

## 4.2 Cenários de Teste

Você deve implementar dois cenários de testes distintos, conforme especificação a seguir. Cada cenário deve ser implementado em um arquivo de teste (arquivo .java) distinto.

### Cenário 1

No cenário 1, você deve criar fakes para simular serviços externos definidos pelas interfaces `IEstoqueExternal` e `IPagamentoExternal`, garantindo que se atinja a meta de cobertura de teste. As outras dependências devem ser implementadas usando mock objects implementados com Mockito.

### Cenário 2

No cenário 2, você deve criar mock objects para simular serviços externos definidos pelas interfaces `IEstoqueExternal` e `IPagamentoExternal`, garantindo que se atinja a meta de cobertura de teste.

Você deve usar fakes para implementar as dependências com a camada `repository`.

---

## 5. Entrega

---

A entrega deve conter:

1. Projeto Maven compactado em formato ZIP.
2. Nome do projeto e `artifactId` do `pom.xml` no formato nome1-nome2.
3. Arquivo README.md contendo:
  - Nome dos autores
  - Instruções de execução
  - Como rodar os testes
  - Como visualizar os relatórios de cobertura
  - Como gerar e interpretar o relatório de mutação

# Apêndice — Prompt para execução do projeto

Copie e cole o prompt abaixo na sua IA (Codex/ChatGPT) para implementar e testar o projeto conforme o enunciado.

Você é um(a) desenvolvedor(a) Java experiente em testes (JUnit 5), Mockito, JaCoCo e PITEST.

Seu objetivo é implementar e testar o trabalho da 3ª unidade de Testes de Software (mutação + dublês), seguindo o enunciado.

REGRAS DO TRABALHO (obrigatório cumprir)

1) Implementar novamente `calcularCustoTotal()` em versão simplificada com estas regras:

- Desconto por valor total do carrinho:
  - \* total  $\geq$  1000,00  $\rightarrow$  20% de desconto
  - \* total  $\geq$  500,00 e  $<$  1000,00  $\rightarrow$  10% de desconto
  - \* demais  $\rightarrow$  sem desconto
- Frete por peso físico total (somente peso físico), por faixa:
  - \* 0-5 kg  $\rightarrow$  frete 0
  - \*  $>5$  e  $\leq 10 \rightarrow$  R\$ 2,00 por kg
  - \*  $>10$  e  $\leq 50 \rightarrow$  R\$ 4,00 por kg
  - \*  $>50 \rightarrow$  R\$ 7,00 por kg
- Produtos frágeis: adicionar R\$ 5,00 POR UNIDADE para cada item marcado como frágil.
- NÃO existe adicional por região nem desconto por fidelidade.
- Ordem de cálculo obrigatória:  
Subtotal (preço unitário \* quantidade, somado)  $\rightarrow$  aplicar desconto  $\rightarrow$  calcular frete  $\rightarrow$  total = subtotalComDesconto + frete

2) Testes obrigatórios de `calcularCustoTotal()`:

- 100% de branch coverage (arestas) no método, comprovado no relatório do JaCoCo.
- 100% de mutantes mortos no método com PITEST (nenhum sobrevivente).
- Documentar no README: comando do PIT, como verificar mutantes sobreviventes, e estratégias usadas para matar mutantes.

3) Testes do método `finalizarCompra()`:

- Cobrir o fluxo completo: verificar estoque (serviço externo)  $\rightarrow$  calcular custo total  $\rightarrow$  autorizar pagamento  $\rightarrow$  atualizar estoque
- 100% de cobertura de decisão (branch/decision) de `finalizarCompra()`.
- Verificar que métodos esperados foram invocados (verify).
- Implementar DOIS CENÁRIOS, cada um em um arquivo de teste .java separado:  
CENÁRIO 1: usar FAKES para `IEstoqueExternal` e `IPagamentoExternal`; demais dependências com Mockito (mocks).  
CENÁRIO 2: usar MOCKS para `IEstoqueExternal` e `IPagamentoExternal`; dependências da camada repository com FAKES.

4) Entrega:

- Projeto Maven ZIP.
- Nome do projeto e artifactId do pom.xml no formato nome1-nome2.
- README.md com: autores, como executar, como rodar testes, como ver cobertura, como gerar/interpretar mutação.

TAREFA (execute sem perguntar, apenas seguindo o repositório atual)

A) Levantamento rápido (no código existente):

1. Localize `calcularCustoTotal()` e `finalizarCompra()` (ou classes equivalentes do e-commerce).
2. Identifique as dependências de `finalizarCompra()` (ex.: `estoqueExternal`, `pagamentoExternal`, repositórios).
3. Liste todos os ramos/decisões existentes em `finalizarCompra()` e quais condições disparam cada ramo.

B) Implementação de `calcularCustoTotal()` (versão simplificada):

1. Ajuste o método para aplicar exatamente as regras acima.
2. Use `BigDecimal` para dinheiro e arredondamento para 2 casas no final (`HALF_UP`, salvo padrão já usado no projeto).
3. Garanta que "frágil" cobra R\$5 por unidade (quantidade do item).
4. Garanta que o frete usa SOMA do peso físico total (`peso * quantidade`, se aplicável).

C) Testes de `calcularCustoTotal()` para 100% BRANCH + 100% MUTATION:

1. Crie uma suíte de testes focada em matar mutantes e cobrir todas as arestas.
2. Inclua testes de fronteira (muito importante para PIT e branch):
  - Desconto: total exatamente 499,99 (sem), 500,00 (10%), 999,99 (10%), 1000,00 (20%).
  - Frete (peso total): 0, 5, 5,01, 10, 10,01, 50, 50,01 (cobrindo todas as faixas).
  - Frágil: zero frágeis e com frágeis (com quantidades  $>1$ ) para validar R\$5/unidade.
  - Arredondamento: pelo menos 1 caso que produza valor com 3+ casas para confirmar arredondamento final.
3. Asserções DEVEM ser precisas (nada de tolerância "delta" em dinheiro). Compare `BigDecimal` com scale 2.
4. Se o projeto já tem JaCoCo configurado, use. Se não tiver, adicione no pom.xml.
5. Se o projeto já tem PIT configurado, use. Se não tiver, adicione plugin PIT no pom.xml e configure `targetClasses`.
6. Objetivo final: relatório JaCoCo mostrando 100% branch no método e PIT com 0 mutantes sobreviventes.

D) Testes de `finalizarCompra()` - dois cenários (2 arquivos .java):

1. Baseie os testes na implementação REAL do método, garantindo 100% de decisão.
2. Cenário 1 (arquivo 1):
  - Implementar `FakeEstoqueExternal` e `FakePagamentoExternal` (classes simples em test/ com comportamento configurável).
  - Outras dependências (ex.: repositórios, serviços auxiliares) como mocks Mockito.
  - Verificar chamadas: `verify(estoqueExternal...)`, `verify(pagamentoExternal...)`, `verify(repo...)`.
  - Garantir pelo menos: caminho de sucesso e pelo menos um caminho de falha (ex.: estoque insuficiente OU pagamento falha).
3. Cenário 2 (arquivo 2):
  - `EstoqueExternal` e `PagamentoExternal` como mocks Mockito (`when/thenReturn`).
  - Repositórios como fakes (implementações em memória) para simular persistência/atualização de estoque e finalizar compra.
  - Novamente: cobrir ramos restantes para fechar 100% de decisão.

4. Para cada teste, valide:

- Resultado retornado (sucesso/falha, mensagem/objeto conforme projeto).
- Efeitos colaterais esperados: estoque atualizado quando sucesso; não atualiza quando falha; pagamento chamado a

E) README.md (obrigatório):

Inclua, no mínimo:

- Autores
- Como executar (mvn clean test)
- Como gerar/ver JaCoCo (onde fica o HTML, ex.: target/site/jacoco/index.html)
- Comando do PIT (ex.: mvn test pitest:mutationCoverage) e onde ver relatório
- Como comprovar que não restaram mutantes sobreviventes (print/trecho do relatório + instrução)
- Estratégias usadas para matar mutantes (ex.: testes de fronteira, assertivas exatas, casos de arredondamento, val

FORMATO DA SUA RESPOSTA (obrigatório)

- 1) Mostre um "plano de mudanças" curto.
- 2) Em seguida, forneça os códigos completos (ou diffs) dos arquivos alterados/criados:
  - implementação de calcularCustoTotal()
  - 1 ou mais arquivos de teste de calcularCustoTotal()
  - TestFinalizarCompraCenario1.java
  - TestFinalizarCompraCenario2.java
  - pom.xml (se alterado)
  - README.md
- 3) Ao final, liste comandos exatos para rodar:
  - testes
  - JaCoCo
  - PITEST

RESTRICÇÕES

- Não invente regras extras além do enunciado.
- Não usar "atalhos" que prejudiquem PIT (asserts frouxos, falta de fronteira).
- Cada cenário de finalizarCompra() DEVE estar em arquivo .java separado.
- artifactId no formato nome1-nome2.