Teste de Software

Implementando Padrões de Teste (Test Patterns)

Aluno: *Lúcio Alves Almeida Neto*

1. Padrões de Criação de Dados (Builders)

1.1. Justificativa: CarrinhoBuilder vs. CarrinhoMother

Neste trabalho, foi implementado o UserMother (Padrão Object Mother) e o CarrinhoBuilder (Padrão Data Builder). A escolha de qual padrão usar foi deliberada e baseada na complexidade da entidade.

O padrão **Object Mother** (UserMother) foi usado para a entidade User porque ela é simples e possui poucas variações relevantes para os testes (Padrão vs. Premium).

No entanto, o **Data Builder** (CarrinhoBuilder) foi escolhido para a entidade Carrinho porque ela é um objeto complexo. Um carrinho pode ter diferentes usuários, uma lista variável de itens, ou pode estar vazio. Se tentássemos usar um CarrinhoMother, teríamos uma "explosão de métodos" para cobrir cada cenário (ex: umCarrinhoPremiumComDoisItens(), umCarrinhoVazio(), umCarrinhoPadraoComItemCaro(), etc.), o que é insustentável.

O CarrinhoBuilder resolve isso permitindo que cada teste configure, de forma fluente e legível, *apenas* o que é importante para aquele cenário específico.

1.2. Exemplo: "Antes" e "Depois"

Para ilustrar a melhoria, veja como seria o "setup" de um teste *sem* o padrão Data Builder ("Antes") e *com* ele ("Depois").

Antes: Setup Manual (Test Smell: "Obscure Setup")

Sem um Builder, o desenvolvedor precisa construir manualmente todos os objetos e suas dependências no "Arrange" do teste. Isso é ruim, difícil de ler e esconde a intenção do teste.

```
// "ANTES" - Setup Manual e Obscuro
test('deve aplicar desconto para premium', async () => {
   // ARRANGE
   // 1. Criar usuário
   const premiumUser = new User(
      'user-premium-456',
      'Usuário Premium',
      'premium@email.com',
      'PREMIUM'
   );
   // 2. Criar itens
   const item1 = new Item('Produto Caro', 200);
   // 3. Criar carrinho com usuário e itens
   const carrinho = new Carrinho(premiumUser, [item1]);

   // ... (Stubs, Mocks, etc.) ...

   // ACT
   // ...
```

});

Depois: Setup com Data Builder

Usando o Builder, o "Arrange" fica limpo, declarativo e focado no que é importante para o teste. O Builder esconde a complexidade de criar itens padrão ou usuários padrão.

```
// "DEPOIS" - Setup Limpo com Builder
test('deve aplicar o desconto e enviar o e-mail', async () => {
   // ARRANGE
   const user = UserMother.umUsuarioPremium();
   const itens = [ new Item('Produto Caro', 200) ];

   // O Builder torna a intenção clara:
   const carrinho = new CarrinhoBuilder()
      .comUser(user)
      .comItens(itens)
      .build();

   // ... (Stubs, Mocks, etc.) ...

   // ACT
   // ...
});
```

1.3. Justificativa da Melhoria

O padrão Data Builder melhora drasticamente a **legibilidade** e a **manutenção** dos testes.

1. Resolve o "Obscure Setup" : O "Antes" é um exemplo claro do Test Smell "Setup Obscuro". O leitor do teste não sabe quais dados (o usuário ser PREMIUM ou o item custar 200) são relevantes e quais são apenas "barulho".
2. **Foco no Cenário:** O "Depois" é explícito. O teste destaca que o usuário *deve* ser premium (.comUser(user)) e os itens *devem* ser os definidos (.comItens(itens)).
3. **Manutenção:** Se a classe Carrinho ganhar um novo campo no construtor (ex: descontoAplicado), em vez de quebrar todos os testes, basta atualizar o .build() do CarrinhoBuilder em um único lugar.

2. Padrões de Test Doubles (Mocks vs. Stubs)

Para esta análise, usei o teste de "sucesso Premium" (Etapa 5) , que testa o cenário describe('quando um cliente Premium finaliza a compra', ...).

Neste teste, várias dependências externas do CheckoutService foram substituídas por "dublês" (Test Doubles).

2.1. Identificação dos Dublês

- **Stub:** O GatewayPagamento e o PedidoRepository foram usados como Stubs.
- **Mock:** O EmailService foi usado como um Mock.

2.2. Justificativa: Stub

O GatewayPagamento foi (principalmente) um **Stub** porque seu papel era *controlar o fluxo do teste*. Para que o CheckoutService pudesse continuar para a lógica de salvar o pedido e enviar o e-mail, ele precisava que o gateway retornasse { success: true }. O teste *preparou* esse retorno fixo para que a execução pudesse prosseguir.

Da mesma forma, o PedidoRepository foi um Stub para fornecer um objeto de pedido salvo e permitir que o código continuasse.

2.3. Justificativa: Mock

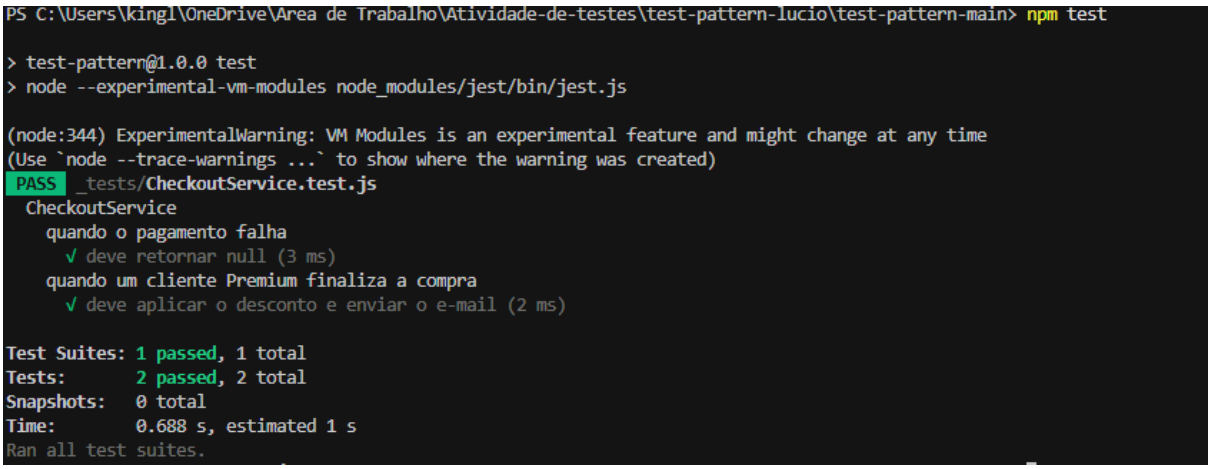
O EmailService foi um **Mock** porque não estávamos interessados no que ele *retornava* (seu estado). O nosso objetivo era verificar *se* e *como* ele foi chamado. As asserções do teste (os expects) focaram 100% no seu comportamento:

1. Ele foi chamado 1 vez? (expect(emailService.enviarEmail).toHaveBeenCalledTimes(1);)
2. Ele foi chamado com os argumentos corretos? (expect(emailService.enviarEmail).toHaveBeenCalledWith('premium@email.com', 'Seu Pedido foi Aprovado!', ...);)

Essa distinção entre verificar o *estado* (o retorno do Stub) e o *comportamento* (as chamadas ao Mock) é a chave para testes de unidade isolados e precisos.

4. Validação Final

A Etapa 6 exigia a execução da suíte de testes completa. Após a implementação de todas as etapas, incluindo a correção dos globais do Jest e dos argumentos de asserção, todos os testes passaram com sucesso.



5. Conclusão

O uso deliberado de Padrões de Teste, como exigido nesta atividade, é fundamental para prevenir "Test Smells" e construir uma suíte de testes sustentável.

Os padrões de criação (**Object Mother** e **Data Builder**) resolveram o problema de "Setup Obscuro", tornando o "Arrange" dos testes limpo, declarativo e de fácil manutenção.

Os padrões de dublês (**Stubs** e **Mocks**) foram essenciais para prevenir "Testes Frágeis". Ao isolar o CheckoutService de suas dependências externas (como o Gateway de Pagamento e o Serviço de E-mail), criamos testes que são rápidos, confiáveis e que testam *apenas* a lógica de negócio do nosso SUT (System Under Test), que era o objetivo final.