Testes de Software

Implementando Padrões de Teste (Test Patterns)

Lucas Ferreira Garcia
804803

1. **Padrões de Criação de Dados (Builders):**

O padrão Data Builder foi utilizado para construir objetos de teste complexos de forma fluente e controlada.
No caso do projeto, o CarrinhoBuilder foi escolhido em vez de um CarrinhoMother, pois o carrinho de compras possui múltiplas variações de itens diferentes, usuários distintos e situações como carrinho vazio ou com vários produtos.
Criar um Object Mother para cada combinação levaria a uma explosão de métodos e setups difíceis de manter.

Antes (manual):

```
const user = new User(1, 'João', 'joao@email.com', 'PADRAO');
const itens = [new Item('Item1', 100), new Item('Item2', 200)];
const carrinho = new Carrinho(user, itens);
```

Depois (CarrinhoBuilder):

```
const carrinho = new CarrinhoBuilder()
  .comUser(UserMother.umUsuarioPremium())
  .comItens([new Item('Item1', 100), new Item('Item2', 200)])
  .build();
```

Resultado:
O Builder melhora a legibilidade e a manutenção dos testes, tornando o setup explícito e eliminando repetições.
Cada cenário de teste mostra apenas o que é relevante, evitando o Test Smell conhecido como Setup Obscuro.

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

O cenário de sucesso Premium (Etapa 5) demonstra o uso combinado e intencional de Stubs e Mocks para isolar dependências e validar o comportamento do sistema. Nesse teste, o objetivo é garantir que um cliente Premium receba o desconto de 10%, que o pagamento seja processado com sucesso, o pedido salvo e o e-mail de confirmação enviado.

As dependências do CheckoutService foram classificadas da seguinte forma:

GatewayPagamento (Stub):
Utilizado para simular o comportamento do serviço externo de cobrança, retornando um resultado controlado:

```
const gatewayStub = { cobrar: jest.fn().mockResolvedValue({ success: true }) };
```

Esse dublê não é usado para verificar interações, mas apenas para garantir que o método processarPedido() receba uma resposta previsível e possa seguir o fluxo de

sucesso.
Por isso, ele é considerado um Stub, pois serve à verificação de estado, ou seja, o foco está no resultado final (pedido processado) e não em como a função foi chamada.

PedidoRepository (Stub):
Também atua como um Stub, retornando um pedido salvo com ID fixo ({ id: 101 }).
Seu papel é apenas fornecer um valor coerente para o fluxo interno do serviço, sem necessidade de verificar chamadas específicas.
Assim, ele contribui para a simulação de dependências persistentes, sem afetar o comportamento da lógica principal.

EmailService (Mock):
Diferente dos anteriores, o EmailService foi usado como um Mock, pois o interesse do teste é verificar o comportamento, isto é, se o método enviarEmail() foi realmente chamado e com os argumentos corretos.

Em resumo, o GatewayPagamento e o PedidoRepository são usados como Stubs, focados em controlar o estado e fluxo da execução, enquanto o EmailService é um Mock, destinado à verificação de interações e efeitos colaterais.
Essa separação clara reforça a intenção do teste e evidencia boas práticas de isolamento, tornando o comportamento do sistema previsível, testável e estável mesmo sem dependências externas reais.

3. **Conclusão:**

O uso deliberado de padrões de teste, como Builders e Test Doubles, previne o surgimento de *Test Smells* e promove uma suíte de testes mais sustentável.

Com Builders, o código de teste torna-se expressivo e fácil de manter.
Com Mocks e Stubs, é possível isolar dependências externas e validar tanto o estado quanto o comportamento do sistema de forma confiável.

Esses padrões aproximam os testes das boas práticas de engenharia de software, garantindo qualidade, clareza e confiabilidade na verificação dos requisitos do sistema.