PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS PUC-MG

TESTE DE SOFTWARE

Nome: Paloma Dias de Carvalho – Matrícula: 587900

Professor: Cleiton Silva Tavares

Relatório – Implementando Padrões de Teste (Test Patterns)

01. Padrões de Criação de Dados (Builders)

01.1. Porque o CarrinhoBuilder foi usado em vez de um CarrinhoMother?

A escolha entre os padrões Builder e Object Mother depende principalmente do nível de flexibilidade e customização exigido na geração de dados de teste. No caso do CarrinhoBuilder, o padrão Builder foi adotado pelos seguintes motivos:

1. Maior Flexibilidade na Criação:
   O Builder possibilita montar objetos com diferentes combinações de atributos por meio de métodos encadeados (fluent interface). Já o Object Mother, em geral, retorna instâncias prontas e fixas, com pouca variação.
2. Customização Gradual:
   Com o Builder, é possível iniciar com valores padrão e alterar apenas o que for necessário para cada cenário de teste específico, tornando o processo mais adaptável e reutilizável.
3. Facilidade de Manutenção:
   Caso a estrutura do objeto Carrinho seja alterada, basta atualizar o Builder em um único ponto, ao invés de modificar diversos métodos dentro de um Object Mother.

A imagem 1 abaixo, apresenta um exemplo de código manual, onde é possível identificar verbose, repetição e baixa manutenabilidade.

```
it('deve aplicar desconto de 10% para cliente Premium', async () => {
  // Cria um usuário do tipo Premium
  const user = new User(2, 'Joao Santos', 'joao.santos@example.com', 'PREMIUM');

  // Cria dois itens de exemplo
  const item1 = new Item('Produto A', 100.00);
  const item2 = new Item('Produto B', 100.00);

  // Adiciona os itens ao carrinho
  const itens = [item1, item2];

  const carrinho = new Carrinho(user, itens);

});
```

Imagem 1 – Exemplo de código

A imagem 2, apresenta o código utilizando builder, de forma mais legível, reutilizável e de fácil manutenção.

```
39          it('deve aplicar desconto de 10% e enviar email de confirmação', async () => {
40              // Arrange
41              const usuarioPremium = UserMother.umUsuarioPremium();
42              const itens = [
43                  new Item('Produto A', 100.00),
44                  new Item('Produto B', 100.00)
45              ];
46              const carrinho = new CarrinhoBuilder()
47                  .comUser(usuarioPremium)
48                  .comItens(itens)
49                  .build();
```

Imagem 2 – Código usando builder

02. Padrões de Test Doubles (Mocks vs Stubs)

Definições:

Stub:
É um *Test Double* usado para retornar valores pré-definidos durante o teste. Seu objetivo principal é verificar o estado, ou seja, confirmar se o resultado final do método testado está correto.

Mock:
É um *Test Double* voltado para verificação de comportamento, garantindo que os métodos foram realmente chamados, com os parâmetros corretos e no número esperado de vezes.

A diferença entre ambos está no foco da verificação:

- Stub: “O que o método retornou?”
- Mock: “O método foi chamado corretamente?”

Identificações:
GatewayPagamento: Stub

O GatewayPagamento foi utilizado principalmente como Stub, pois:

- Retorna um valor pré-configurado { success: true }, essencial para que o teste continue normalmente (sem ele, o método retornaria null).

Apesar disso, o comportamento também foi verificado com toHaveBeenCalledWith para confirmar que o desconto foi aplicado corretamente. Essa verificação é importante porque:

- Garante que o valor enviado ao gateway foi de R$ 180,00 (aplicando 10% de desconto sobre R$ 200,00).
- Confirma que a regra de desconto para clientes Premium está funcionando como esperado.

Em resumo, ele é principalmente um Stub, pois o foco está em simular uma resposta esperada e permitir o fluxo do teste. A verificação de comportamento é secundária, servindo apenas como apoio para validar a lógica do desconto.

O EmailService foi usado como Mock porque:

- O retorno do método não é relevante (undefined).
- O objetivo é verificar se o método foi chamado e com quais parâmetros.
- É necessário garantir que o e-mail de confirmação foi enviado corretamente após um pagamento bem-sucedido.

O foco aqui está no comportamento, não no valor de retorno. O envio do e-mail é um efeito colateral importante do processo de checkout que precisa ser validado, e por isso o EmailService se enquadra como Mock.

O EmailService foi usado como Mock porque:

- O retorno do método não é relevante (undefined).
- O objetivo é verificar se o método foi chamado e com quais parâmetros.
- É necessário garantir que o e-mail de confirmação foi enviado corretamente após um pagamento bem-sucedido.

O foco aqui está no comportamento, não no valor de retorno. O envio do e-mail é um efeito colateral importante do processo de checkout que precisa ser validado, e por isso o EmailService se enquadra como Mock.

O PedidoRepository foi usado como Stub porque:

- Retorna um valor pré-definido (por exemplo, o pedido salvo com um ID).
- É necessário para que o teste prossiga (o e-mail precisa do ID do pedido).
- Seu comportamento não é verificado, apenas o resultado.

Portanto, ele é considerado um Stub, já que o foco está no resultado retornado (pedido salvo), e não em como o método foi chamado. O repositório atua apenas como uma dependência de apoio, sem impacto direto na verificação principal do teste.

03. Conclusão

A utilização de diferentes tipos de *Test Doubles*, como Stubs e Mocks, foi essencial para garantir que os testes cobrissem tanto o estado quanto o comportamento das unidades testadas. Enquanto os *Stubs* permitiram simular respostas e manter o fluxo de execução sem dependências externas reais, os *Mocks* foram fundamentais para validar as interações entre os componentes, assegurando que métodos fossem chamados corretamente e com os parâmetros esperados.

Essa combinação proporcionou testes mais confiáveis, claros e de fácil manutenção, possibilitando a verificação completa das regras de negócio, como o desconto para clientes *Premium* e o envio de e-mails após o pagamento. Em resumo, o uso equilibrado entre Stubs e Mocks tornou os testes mais expressivos, evitando falsos positivos e garantindo a consistência do comportamento do sistema.