Teste de Software

# *Test Patterns*

Arthur Henrique Oliveira Assunção
787869

1. Padrões de Criação de Dados (Builders)

1.1. Escolha do Padrão

Durante o desenvolvimento da suíte de testes, foram aplicados dois padrões para criação de dados: Object Mother e Data Builder.
O Object Mother foi utilizado para criar objetos simples e estáticos, como usuários padrão e premium, enquanto o Data Builder foi escolhido para objetos mais complexos e variáveis — neste caso, o Carrinho de Compras.

O CarrinhoBuilder foi usado em vez de um *CarrinhoMother* porque o carrinho possui múltiplas variações de configuração (itens, usuários, valores e estados).
Se usássemos um Object Mother, seria necessário criar vários métodos como umCarrinhoComUmItem(), umCarrinhoVazio(), umCarrinhoPremiumComDoisItens(), etc., resultando em código redundante e de difícil manutenção.
O Builder, por outro lado, permite criar objetos de forma fluente e customizável, mantendo o código limpo e legível.

1.2. Exemplo “Antes” (Setup Manual Complexo)

```
const usuario = { nome: "Usuário Premium", tipo: "PREMIUM", email: "premium@email.com" };
const carrinho = {
  usuario,
  itens: [{ nome: "Produto Especial", preco: 200 }],
  getTotal: () => 200,
};
```

1.3. Exemplo “Depois” (Usando o Builder)

```
const usuario = UserMother.umUsuarioPremium();
const carrinho = new CarrinhoBuilder()
  .comUsuario(usuario)
  .comItens([{ nome: "Produto Especial", preco: 200 }])
  .build();
```

1.4. Benefícios

O uso do Data Builder melhora a legibilidade, pois expressa a intenção do teste de forma natural (ex: “criar carrinho com usuário Premium e itens personalizados”).
Além disso, facilita a manutenção, pois qualquer mudança na estrutura do carrinho é feita apenas dentro do builder, sem afetar os testes que o utilizam.
Esse padrão também elimina o *Test Smell* de *Setup Obscuro*, tornando o contexto do teste claro e explícito.

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

2.1. Contexto

Nos testes do CheckoutService, foram aplicados os conceitos de Stubs e Mocks para isolar dependências externas, garantindo que os testes validem apenas a

lógica de negócio.

O cenário de sucesso com um cliente Premium exemplifica bem essa diferença.

2.2. Dependências e Funções

- GatewayPagamento → Stub:

  Foi usado para controlar o fluxo do teste, simulando a resposta do serviço de pagamento.

  Ele retorna { success: true } ou { success: false } conforme o cenário.

  O foco está na verificação de estado (se o pedido foi processado com sucesso ou não).

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

- EmailService → Mock:

  Foi usado para verificar comportamento, isto é, se o método enviarEmail foi chamado corretamente e com os argumentos esperados.

  O foco está em como o serviço foi chamado, não em seu retorno.

```
const emailMock = { enviarEmail: jest.fn() };
expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
expect(emailMock.enviarEmail).toHaveBeenCalledWith(
  "premium@email.com",
  "Seu Pedido foi Aprovado!",
  expect.anything()
);
```

3. Conclusão

O uso deliberado de Padrões de Teste — como *Object Mother*, *Data Builder*, *Stubs* e *Mocks* — transforma completamente a qualidade e sustentabilidade da suíte de testes.

Esses padrões preveem Test Smells como setups obscuros, duplicação de código e dependências externas reais.
Além disso, tornam os testes mais expressivos, rápidos e confiáveis, facilitando a leitura e compreensão até por quem não conhece a implementação interna do sistema.

Em resumo:

- Builders reduzem a complexidade de setup.
- Doubles (Mocks/Stubs) garantem isolamento e clareza de intenção.
- O resultado é uma suíte de testes limpa, robusta e sustentável, alinhada às boas práticas de engenharia de software moderna.