Teste de Software

Análise e Implementação de Padrões de Teste: Builders e Test Doubles

João Pedro Silva Braga
815051

Belo Horizonte, MG
9 de novembro de 2025

1 Padrões de Criação de Dados (Builders)

A criação de dados para testes é um pilar para a clareza e manutenibilidade de uma suíte de testes. Um *setup* de teste confuso, conhecido como o *Test Smell* de "Setup Obscuro", pode esconder a intenção real do teste e torná-lo frágil. Neste projeto, enfrentamos este desafio ao modelar os objetos de domínio `User` e `Carrinho`, utilizando duas abordagens complementares: **Object Mother** e **Data Builder**.

1.1 A Escolha Estratégica: Data Builder para Carrinho

A decisão entre os padrões Object Mother e Data Builder depende diretamente da complexidade e variabilidade do objeto em questão. Para a entidade `User`, que possui poucos estados fixos e bem definidos (ex: "Padrão"e "Premium"), o padrão **Object Mother** é uma solução elegante e direta, oferecendo instâncias prontas para uso com um simples chamado de método estático.

No entanto, o `Carrinho` é uma entidade muito mais dinâmica. Ele pode conter um número variável de itens, com diferentes preços, pertencer a diferentes usuários ou até mesmo estar vazio. Tentar modelar todas essas variações com um Object Mother levaria a uma "explosão combinatória"de métodos, como `carrinhoComUmItem()`, `carrinhoVazio()`, etc. Essa abordagem não escala e torna a manutenção da fábrica de testes um desafio por si só.

O padrão **Data Builder**, por outro lado, foi projetado para lidar com essa complexidade. Através de uma interface fluente, ele nos permite construir um objeto `Carrinho` passo a passo, de forma declarativa e explícita dentro do próprio teste. Isso torna a intenção do cenário de teste cristalina, focando apenas nos atributos que são relevantes para aquele caso específico.

1.2 Análise Comparativa: Setup Manual vs. Setup com Builders

Para ilustrar o impacto do Data Builder, comparemos a criação de um cenário de teste antes e depois da sua implementação.

Antes (Setup Manual): Sem um padrão de criação, a fase de *Arrange* do teste fica poluída com detalhes de instanciação, o que desvia a atenção da lógica que está sendo validada.

```
import { CheckoutService } from '../src/services/CheckoutService.
    js';
import { Carrinho } from '../src/domain/Carrinho.js';
import { Item } from '../src/domain/Item.js';
import { User } from '../src/domain/User.js';

test('checkout p/ usuário padrão (setup manual)', async () => {
  const usuario = new User(1, 'Usuário Padrão', 'padrao@example.
     com', 'PADRAO');
  const itens = [ new Item('Produto1', 30), new Item('Produto2',
     20) ];
  const carrinho = new Carrinho(usuario, itens);

  const gatewayStub = { cobrar: jest.fn().mockResolvedValue({
     success: true }) };
  const pedidoSalvoFake = { id: 99, carrinho, totalFinal: 50,
     status: 'PROCESSADO' };
  const repoStub = { salvar: jest.fn().mockResolvedValue(
     pedidoSalvoFake) };
  const emailMock = { enviarEmail: jest.fn().mockResolvedValue(
     true) };

  const service = new CheckoutService(gatewayStub, repoStub,
     emailMock);
  const resultado = await service.processarPedido(carrinho, {
     numero: '123' });

  expect(resultado).toEqual(pedidoSalvoFake);
  expect(gatewayStub.cobrar).toHaveBeenCalledWith(50, { numero: '
     123' });
});
```

Listing 1: Setup manual sem Builder

Depois (Com Data Builder e Object Mother): A combinação dos padrões transforma a fase de *Arrange*, tornando-a concisa, legível e expressiva. O código passa a contar uma história clara sobre o cenário em preparação.

```
import { CheckoutService } from '../src/services/CheckoutService.
    js';
import { CarrinhoBuilder } from './builders/CarrinhoBuilder.js';
import { UserMother } from './builders/UserMother.js';

test('checkout p/ usuário padrão (com Builder)', async () => {
  const usuario = UserMother.padrao();
  const carrinho = new CarrinhoBuilder()
    .comUsuario(usuario)
    .adicionarItem('Produto1', 30)
    .adicionarItem('Produto2', 20)
    .build();

  const gatewayStub = { cobrar: jest.fn().mockResolvedValue({
     success: true }) };
  const pedidoSalvoFake = { id: 99, carrinho, totalFinal: 50,
     status: 'PROCESSADO' };
  const repoStub = { salvar: jest.fn().mockResolvedValue(
     pedidoSalvoFake) };
  const emailMock = { enviarEmail: jest.fn().mockResolvedValue(
     true) };

  const service = new CheckoutService(gatewayStub, repoStub,
     emailMock);
  const resultado = await service.processarPedido(carrinho, {
     numero: '123' });

  expect(resultado).toEqual(pedidoSalvoFake);
  expect(gatewayStub.cobrar).toHaveBeenCalledWith(50, { numero: '
     123' });
});
```

Listing 2: Setup com CarrinhoBuilder e UserMother

A transformação é evidente. O *setup* no segundo exemplo é mais do que apenas código; ele narra o cenário: "Dado um usuário padrão e um carrinho construído com estes itens específicos...". Isso elimina o "ruído"da instanciação manual, permitindo que qualquer leitor compreenda o contexto do teste rapidamente. A manutenção também é simplificada: adicionar ou modificar um item torna-se uma questão de adicionar ou alterar uma única linha na interface fluente, sem impactar outros testes.

2 Padrões de Test Doubles (Mocks vs. Stubs)

Para garantir que nossos testes de unidade sejam rápidos, determinísticos e verdadeiramente isolados, é crucial substituir dependências externas como gateways de pagamento e serviços de e-mail por **Test Doubles**. No entanto, nem todos os dublês são iguais. A distinção entre **Stubs** e **Mocks** é fundamental, pois está ligada ao que desejamos verificar: o **estado** resultante da operação ou o **comportamento** do nosso sistema durante a execução.

2.1 Análise do Cenário: Checkout de Cliente Premium

O cenário de teste para um usuário Premium foi escolhido por sua riqueza de interações. Ele não apenas envolve um fluxo de sucesso, mas também uma lógica de negócio específica (a aplicação de um desconto de 10%) e um efeito colateral (o envio de um e-mail de confirmação). Isso o torna o exemplo perfeito para dissecar a diferença entre Stubs e Mocks na prática.

```
// Arrange
const usuario = UserMother.premium();
const carrinho = new CarrinhoBuilder().comUsuario(usuario).
    comTotalDe(200.00).build();

const gatewayStub = { cobrar: jest.fn().mockResolvedValue({
    success: true }) };
const repoStub = { salvar: jest.fn().mockResolvedValue({ id: 1,
    ... }) };
const emailMock = { enviarEmail: jest.fn() };

const service = new CheckoutService(gatewayStub, repoStub,
    emailMock);

// Act
await service.processarPedido(carrinho, { numero: '999' });

// Assert
expect(gatewayStub.cobrar).toHaveBeenCalledWith(180, expect.any(
    Object));
expect(emailMock.enviarEmail).toHaveBeenCalledWith('premium@email
    .com', 'Seu Pedido foi Aprovado!');
```

Listing 3: Trecho do teste de sucesso para cliente Premium.

2.2 Identificando os Papéis: Stub vs. Mock

Neste teste, cada dublê desempenha um papel distinto:

- **GatewayPagamento (Predominantemente um Stub):** A principal função do gateway na maioria dos testes é controlar o fluxo da execução. Ele é um **Stub** quando o configuramos para retornar `{ success: true }` ou `{ success: false }`. Com isso, forçamos o Sistema Sob Teste (SUT) a seguir o caminho de sucesso ou de falha. A verificação se concentra no **estado** final retornado pelo método (um pedido ou nulo). No entanto, neste teste específico, ele também assume um papel de **Mock** quando verificamos *com quais argumentos* o método `cobrar` foi chamado (`toHaveBeenCalledWith(180, ...)`). Essa é uma verificação de interação, ou seja, de **comportamento**.

- **PedidoRepository (Puro Stub):** Este dublê atua puramente como um **Stub**. Sua única responsabilidade é fornecer um objeto de pedido salvo quando o método `salvar` é invocado. Não estamos interessados em como ou quantas vezes ele foi chamado, apenas no valor que ele retorna para que o SUT possa completar seu fluxo e nós possamos validar o estado final.

- **EmailService (Puro Mock):** É o exemplo canônico de um **Mock**. O envio de um e-mail é uma ação de "dispare e esqueça"do ponto de vista do SUT; ele não retorna um valor que altera a lógica de negócio. Portanto, a única maneira de garantir que o sistema se comportou corretamente é verificar se a interação (a chamada a `enviarEmail`) ocorreu, e com os argumentos exatos. Esta é a essência da **verificação de comportamento**.

A distinção é crucial: usamos Stubs para fornecer dados e controlar o fluxo (verificação de estado), e usamos Mocks para validar que interações e efeitos colaterais importantes aconteceram como esperado (verificação de comportamento).

3 Conclusão

A aplicação disciplinada de Padrões de Teste como **Data Builders**, **Object Mothers**, **Stubs** e **Mocks** representa uma mudança de paradigma: passamos da correção reativa de *Test Smells* para a sua prevenção proativa. Ao construir testes limpos desde o início, garantimos que a suíte de testes seja um ativo sustentável para o projeto.

O **Data Builder** atacou diretamente os *Test Smells* de `Obscure Test` e `General Fixture`, tornando a fase de *Arrange* explícita e relevante para cada cenário. A utilização de **Test Doubles** quebrou o acoplamento com dependências externas, prevenindo `Brittle Tests` (testes frágeis) e garantindo que os testes de unidade sejam rápidos e determinísticos.

Finalmente, a distinção clara entre Stubs (para verificação de estado) e Mocks (para verificação de comportamento) nos permitiu escrever asserções mais focadas e significativas. O resultado é uma suíte de testes que não apenas detecta regressões, mas também serve como uma documentação viva, é resiliente a mudanças no código de produção e inspira confiança na qualidade do software.