PONTÍFICIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Engenharia de Software – Teste de Software

Aluno: Gabriel Lourenço Reis Resende

Professor: Cleiton Silva Tavares

Implementando Padrões de Teste (Test Patterns)

Belo Horizonte

2025

1. Introdução

O desenvolvimento de software de qualidade depende não apenas da implementação correta das funcionalidades, mas também da construção de uma suíte de testes robusta e sustentável. Testes bem projetados garantem que mudanças futuras não quebrem funcionalidades existentes e permitem refatorações seguras. No entanto, testes mal estruturados podem se tornar difíceis de entender, manter e evoluir, devido aos *Test Smells*. Neste relatório, apresentamos como a aplicação de Padrões de Criação de Dados (*Builders*) e Padrões de *Test Doubles* (*Stubs e Mocks*) auxilia na prevenção de *Test Smells*, proporcionando testes claros, legíveis e isolados. Foi utilizado como contexto um serviço de checkout de *e-commerce*, onde validamos cenários de falha e sucesso de pagamento, incluindo clientes premium que recebem desconto.

Repositório do projeto: https://github.com/gabrielreisresende/test-pattern

2. Padrões de Criação de Dados (*Builders*)

No desenvolvimento de testes, criar objetos complexos de forma legível e flexível é um desafio constante. O Carrinho, por exemplo, contém um usuário, múltiplos itens e valores monetários, podendo variar de acordo com o cenário de teste. Criar instâncias manualmente em cada teste gera setup obscuro, tornando os testes difíceis de ler e manter.

Para resolver isso, utilizamos o padrão *Data Builder*, implementado no *CarrinhoBuilder*. Diferente de um *Object Mother* tradicional, que geraria métodos fixos como umCarrinhoComUmItem() ou umCarrinhoVazio(), o *Builder* permite configurar fluentemente apenas os atributos relevantes para cada teste, evitando a explosão de métodos e aumentando a flexibilidade.

Cenário com s*etup* complexo sem utilização do *Builder*:

```
const usuario = new User('Gabriel', 'premium@email.com', true);
const item1 = new Item('Produto 1', 100);
const item2 = new Item('Produto 2', 100);
const carrinho = new Carrinho(usuario, [item1, item2]);
```

Cenário após implementação do Builder:

O Builder deixa claro quais atributos são relevantes para aquele teste, com um setup legível e facilmente ajustável. Se o teste precisar de um carrinho vazio ou com apenas um item, basta alterar o método .comItens() ou chamar .vazio().

```
const carrinho = new CarrinhoBuilder()
    .comUser(usuarioPremium)
    .comItens([
        { nome: 'Produto 1', preco: 100 },
        { nome: 'Produto 2', preco: 100 }
    ])
    .build();
```

3. Padrões de *Test Doubles (Mocks vs. Stubs)*

No teste de sucesso para um cliente *premium* (Etapa 5), identificamos as dependências externas do *CheckoutService*:

- *GatewayPagamento* → *Stub*
- *PedidoRepository* → *Stub*
- *EmailService* → *Mock*

Stubs controlam o retorno de uma dependência para validar o estado final do sistema. O *GatewayPagamento* é um *Stub*, retornando *{ success: true }* para simular pagamento aprovado, permitindo que o teste valide que o desconto de 10% foi aplicado corretamente e que o pedido final retornado está correto.

O *EmailService*, por outro lado, é um *Mock*, porque precisamos validar interações, ou seja, que o e-mail foi enviado uma vez, para o usuário correto, com o assunto e corpo esperados. Aqui a verificação é de comportamento, não do estado interno do pedido.

```
describe('quando um cliente Premium finaliza a compra', () => {
    test('deve aplicar desconto e enviar e-mail', async () => {
        // Arrange
        const usuarioPremium = UserMother.umUsuarioPremium();

        // Carrinho com R$ 200,00 em itens
        const carrinho = new CarrinhoBuilder()
            .comUser(usuarioPremium)
            .comItens([
                { nome: 'Produto 1', preco: 100 },
                { nome: 'Produto 2', preco: 100 }
            ])
            .build();

        const cartaoCredito = '1234-5678-9012-3456';

        const gatewayStub = { cobrar: jest.fn().mockResolvedValue({ success: true }) };
        const pedidoSalvoMock = { id: 1, carrinho, total: 180, status: 'PROCESSADO' };
        const pedidoRepositoryStub = { salvar: jest.fn().mockResolvedValue(pedidoSalvoMock) };
        const emailMock = { enviarEmail: jest.fn().mockResolvedValue(true) };

        const checkoutService = new CheckoutService(
            gatewayStub,
            pedidoRepositoryStub,
            emailMock
        );

        // Act
        const pedidoFinal = await checkoutService.processarPedido(carrinho, cartaoCredito);

        // Assert
        expect(pedidoFinal).toEqual(pedidoSalvoMock);
        expect(gatewayStub.cobrar).toHaveBeenCalledWith(180, cartaoCredito);
        expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
        expect(emailMock.enviarEmail).toHaveBeenCalledWith(
            usuarioPremium.email,
            'Seu Pedido foi Aprovado!',
            `Pedido ${pedidoSalvoMock.id} no valor de R$180`
        );
    });
});
});
```

4. Conclusão

O uso de padrões de teste, como *Builders* e *Test Doubles*, demonstra-se essencial para a construção de uma suíte de testes robusta e sustentável. O *Data Builder*, por exemplo, permite criar objetos complexos de forma legível e flexível, tornando explícito apenas o que é relevante para cada cenário de teste e evitando setups obscuros ou redundantes. Já os *Test Doubles — Stubs e Mocks —* garantem que dependências externas não interfiram nos resultados dos testes, permitindo tanto a verificação de estado quanto a verificação de comportamento, de acordo com o objetivo do teste.

Dessa forma, a aplicação consciente desses padrões previne a ocorrência de *Test Smells,* aumenta a clareza e a manutenção dos testes, e proporciona maior confiança na evolução do sistema. Testes bem estruturados e isolados possibilitam identificar falhas de forma mais rápida e segura, incentivam a refatoração contínua e contribuem para que a suíte de testes funcione como uma ferramenta confiável de garantia de qualidade do *software*, tornando o desenvolvimento mais eficiente e seguro.