Pontifícia Universidade Católica de Minas Gerais

Engenharia de Software

Implementando Padrões de Teste

Test Patterns no Serviço de Checkout

Gabriel Henrique

Matrícula: 814050

Belo Horizonte

Novembro de 2025

1 Padrões de Criação de Dados (Builders)

1.1 Justificativa para uso do CarrinhoBuilder

Durante a implementação dos testes, percebi que o padrão Object Mother funcionou perfeitamente para a classe `User`, já que os usuários têm configurações simples e fixas (padrão ou premium). Criei o `UserMother` com apenas dois métodos: `umUsuarioPadrao()` e `umUsuarioPremium()`.

Porém, ao começar a testar o `Carrinho`, vi que esse padrão não seria adequado. O carrinho é um objeto complexo que varia muito entre os testes: pode estar vazio, ter um item, ter vários itens, ter usuário padrão ou premium, ter valores diferentes. Se eu usasse Object Mother aqui, teria que criar métodos como:

- `umCarrinhoVazio()`
- `umCarrinhoComUmItem()`
- `umCarrinhoComDoisItens()`
- `umCarrinhoComUsuarioPremium()`
- `umCarrinhoComUsuarioPremiumEDoisItens()`

Isso levaria a uma explosão de métodos e ainda assim não cobriria todas as combinações possíveis. Por isso, escolhi implementar o padrão Data Builder para o carrinho.

1.2 Comparação: Antes e Depois

Antes de usar o Builder, o setup dos testes ficava assim:

Listing 1: Setup sem Builder

```
it('deve␣aplicar␣desconto␣premium', async () => {
    const user = new User(2, 'Maria␣Santos',
                          'premium@email.com', 'PREMIUM');
    const item1 = new Item('Notebook', 150);
    const item2 = new Item('Mouse', 50);
    const itens = [item1, item2];
    const carrinho = new Carrinho(user, itens);

    // resto do teste...
});
```

O problema aqui é que tem muita informação desnecessária (ID do usuário, nome completo) e fica difícil de identificar rapidamente o que é importante para esse teste específico.

Depois de implementar o Builder, o mesmo setup ficou assim:

Listing 2: Setup com Builder

```
it('deve␣aplicar␣desconto␣premium', async () => {
    const carrinho = new CarrinhoBuilder()
        .comUser(UserMother.umUsuarioPremium())
        .comItens([
            new Item('Notebook', 150),
            new Item('Mouse', 50)
        ])
        .build();

    // resto do teste...
});
```

A diferença é clara: agora fica explícito que o teste é sobre um usuário premium com R$ 200 em itens. Só o que importa está visível.

O Builder também facilitou muito a criação de outros cenários. Por exemplo, para testar um carrinho vazio, basta fazer:

```
const carrinho = new CarrinhoBuilder().vazio().build();
```

Isso melhora a legibilidade dos testes e facilita a manutenção, pois se eu precisar mudar a estrutura do Carrinho no futuro, só preciso atualizar o Builder, não todos os testes.

2 Padrões de Test Doubles (Mocks vs Stubs)

2.1 Análise do teste Premium

No teste que verifica o fluxo completo para um cliente premium, eu precisei isolar três dependências externas: o gateway de pagamento, o repositório e o serviço de email. Cada uma teve um papel diferente.

2.2 GatewayPagamento como Stub

O gateway de pagamento foi usado como Stub porque meu objetivo era controlar o retorno dele para permitir que o teste continuasse. Configurei assim:

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

Depois, no assert, verifiquei se o gateway foi chamado com o valor correto:

```
expect(gatewayStub.cobrar).toHaveBeenCalledWith(180, '
   1234-5678-9012-3456');
```

O que eu estava testando aqui era o **estado**: o valor que chegou no gateway estava correto após aplicar o desconto de 10%? (R$ 200 - 10% = R$ 180). O gateway é um Stub porque o foco está no resultado da operação, não na interação em si.

2.3 EmailService como Mock

Já o serviço de email foi usado como Mock porque o que importava era verificar se a interação aconteceu corretamente:

```
const emailMock = {
    enviarEmail: jest.fn().mockResolvedValue(true)
};
```

No assert, verifiquei não só se o método foi chamado, mas quantas vezes e com quais argumentos:

```
expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
expect(emailMock.enviarEmail).toHaveBeenCalledWith(
    'premium@email.com',
    'Seu Pedido foi Aprovado!',
    'Pedido PED-123 no valor de R$180'
);
```

Aqui o foco está no **comportamento**: o sistema enviou o email? Enviou só uma vez? Com os dados corretos? Isso é verificação de comportamento, por isso é um Mock.

A diferença principal é que se eu remover as verificações do Mock, o teste falha, porque o comportamento esperado não foi validado. Já se eu remover as verificações do Stub, o teste pode continuar passando, pois o Stub existe apenas para permitir que o código continue executando.

3 Conclusão

A implementação desses padrões de teste me fez entender na prática a diferença entre corrigir problemas (como fazíamos refatorando Test Smells) e prevenir que eles apareçam desde o início.

O uso do Builder evitou que os testes ficassem com setup obscuro e difícil de entender. Cada teste agora deixa claro quais são as condições específicas que está testando.

Já o uso correto de Stubs e Mocks garantiu que os testes ficassem isolados das dependências externas. Isso torna os testes mais rápidos, confiáveis e fáceis de manter. Além disso, entendi que Stubs focam em verificar estados enquanto Mocks focam em verificar comportamentos - conceitos diferentes que resolvem problemas diferentes.

No geral, os padrões de teste funcionam como uma forma de documentação do sistema. Quando alguém lê os testes que escrevi, consegue entender rapidamente o que cada cenário faz sem precisar decifrar código complexo. Isso contribui para uma suíte de testes sustentável a longo prazo.

Referências

1. MESZAROS, Gerard. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

2. FOWLER, Martin. *Mocks Aren't Stubs*, 2007. Disponível em: https://martinfowler.com/article

3. Jest Documentation. *Mock Functions*. Disponível em: https://jestjs.io/docs/mock-functions