Pontifícia Universidade Católica de Minas Gerais

Engenharia de Software

Padrões de Teste

Aplicação de Builders e Test Doubles

Luiz Filipe Nery Costa

Matrícula: 805836

Testes de Software

Belo Horizonte
2025

1 Padrões de Criação de Dados

1.1 Por que CarrinhoBuilder ao invés de CarrinhoMother?

O padrão Object Mother funciona bem para entidades simples, mas o Carrinho é um objeto complexo que pode ter muitas combinações: diferentes usuários, vários itens, ou estar vazio. Usar um CarrinhoMother levaria a uma explosão de métodos como `umCarrinhoVazio()`, `umCarrinhoCom2Itens()`, `umCarrinhoComUsuarioPremium()`, etc.

O CarrinhoBuilder resolve este problema oferecendo uma API fluente que permite customizar apenas o que é relevante para cada teste, mantendo valores padrão sensatos para o resto.

1.2 Exemplo: Antes e Depois

Antes (Setup Manual):

```
const usuario = new User(2, 'Maria Santos', 'premium@email.com', '
    PREMIUM');
const item1 = new Item('Mouse', 100);
const item2 = new Item('Teclado', 100);
const carrinho = new Carrinho(usuario, [item1, item2]);
```

Depois (Com Builder):

```
const usuarioPremium = UserMother.umUsuarioPremium();
const carrinho = new CarrinhoBuilder()
    .comUser(usuarioPremium)
    .comItens([new Item('Mouse', 100), new Item('Teclado', 100)])
    .build();
```

1.3 Como o Builder Melhora

Legibilidade: A sintaxe fluente deixa claro o que está sendo configurado. Métodos como `.comUser()` e `.comItens()` são auto-explicativos.

Manutenção: Se o construtor do Carrinho mudar (novos parâmetros, ordem diferente), apenas o Builder precisa ser atualizado. Os testes permanecem inalterados.

Flexibilidade: Cada teste configura apenas o necessário. Um teste sobre carrinho vazio usa `.vazio()`, enquanto um teste sobre desconto premium especifica o usuário e itens.

2 Padrões de Test Doubles

2.1 Identificação no Teste de Sucesso Premium

No teste onde um cliente premium finaliza a compra, três dependências foram configuradas:

Stub: GatewayPagamento

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

Stub: PedidoRepository

```
const repositoryStub = {
    salvar: jest.fn().mockResolvedValue(pedidoSalvo)
};
```

Mock: EmailService

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

2.2 Por que Gateway é Stub e Email é Mock?

GatewayPagamento como Stub:

O gateway foi usado principalmente para controlar o fluxo do teste. Configuramos seu retorno (`success: true`) para permitir que o checkout prossiga. Embora tenhamos verificado se foi chamado com o valor correto (R$ 180 com desconto), o foco principal foi garantir que o desconto foi aplicado corretamente. A verificação aqui é de **estado**: o valor final calculado está correto?

EmailService como Mock:

O envio de e-mail é um efeito colateral crítico do sistema. Não basta saber que o checkout funcionou, precisamos garantir que o cliente foi notificado. Por isso, verificamos explicitamente se o `enviarEmail` foi chamado, quantas vezes foi chamado e com quais argumentos. A verificação aqui é de **comportamento**: o sistema se comunicou corretamente com o serviço externo?

```
expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
expect(emailMock.enviarEmail).toHaveBeenCalledWith(
    'premium@email.com',
    'Seu Pedido foi Aprovado!',
    'Pedido PED-001 no valor de R$180'
);
```

2.3 Verificação de Estado vs. Comportamento

A diferença fundamental entre Stub e Mock está no tipo de verificação:

Stubs controlam o que dependências retornam, permitindo testar diferentes caminhos do código. Verificamos o **estado** resultante da operação.

Mocks não apenas controlam retornos, mas também registram interações. Verificamos o **comportamento**: o sistema chamou as dependências corretas? Com os argumentos certos? Na ordem esperada?

No contexto do checkout, cobrar no gateway é parte do fluxo (estado), enquanto notificar o cliente é uma responsabilidade a ser verificada (comportamento).

3 Conclusão

O uso deliberado de padrões de teste trouxe benefícios práticos ao projeto. O CarrinhoBuilder eliminou o test smell de "setup obscuro", tornando os testes mais legíveis ao explicitar apenas o que é relevante. A manutenção foi facilitada ao isolar mudanças na construção de objetos em um único lugar.

A distinção entre Stubs e Mocks permitiu criar testes focados e menos frágeis. Stubs controlam o fluxo sem poluir os testes com verificações desnecessárias, enquanto Mocks garantem que comportamentos críticos, como notificações, ocorram corretamente.

Esses padrões previnem test smells comuns: setup obscuro, dados mágicos espalhados, testes frágeis que quebram com qualquer mudança, e verificação excessiva de interações irrelevantes. O resultado é uma suíte de testes que serve como documentação viva do sistema, confiável e sustentável ao longo da evolução do código.