Padrões de Teste: Builders e Test Doubles

Disciplina: Testes de Software
Trabalho: Implementação de Padrões de Teste (Test Patterns)
Aluno: Ana Carolina Caldas de Mello
Matrícula: 801198
Data: Novembro de 2024

---

1. Padrões de Criação de Dados (Builders)

1.1 Por que CarrinhoBuilder em vez de CarrinhoMother?

A escolha entre **Builder** e **Object Mother** depende das necessidades de flexibilidade e customização dos dados de teste. No caso do `CarrinhoBuilder`,o padrão **Builder** foi escolhido pelas seguintes razões:

1. **Flexibilidade na Construção**: O Builder permite criar objetos com diferentes combinações de propriedades através de métodos fluentes, enquanto um Object Mother geralmente retorna objetos pré-configurados fixos.

2. **Customização Incremental**: Com o Builder, é possível começar com valores padrão e customizar apenas o que é necessário para cada teste específico.

3. **Manutenção**: Se a estrutura do `Carrinho` mudar, é necessário atualizar apenas o Builder, não múltiplos métodos do Object Mother.

Por outro lado, o `UserMother` foi implementado como Object Mother porque usuários têm variações bem definidas e limitadas (padrão, premium), não necessitando de tanta flexibilidade na construção.

1.2 Exemplo: Antes e Depois

Antes: Setup Manual Complexo

```
it('deve aplicar desconto de 10% para cliente Premium', async () => {
    const user = new User(2, 'Jane Doe', 'jane.doe@example.com', 'PREMIUM');
    const item1 = new Item('Produto A', 100.00);
    const item2 = new Item('Produto B', 100.00);
    const itens = [item1, item2];
    const carrinho = new Carrinho(user, itens);
});
```

Problemas do código:
- **Verboso**: Muitas linhas para criar um objeto simples
- **Repetitivo**: Código duplicado em múltiplos testes

- **Difícil de manter**: Mudanças na estrutura do `Carrinho` exigem atualização em vários lugares

Depois: Usando o Builder

```
it('deve aplicar desconto de 10% para cliente Premium', async () => {
    const usuarioPremium = UserMother.umUsuarioPremium();
    const itens = [
        new Item('Produto A', 100.00),
        new Item('Produto B', 100.00)
    ];
    const carrinho = new CarrinhoBuilder()
        .comUser(usuarioPremium)
        .comItens(itens)
        .build();
});
```

Benefícios do código :
- **Conciso**: Menos linhas, mais legível
- **Reutilizável**: O Builder pode ser usado em múltiplos testes com diferentes configurações
- **Manutenível**: Mudanças na estrutura do `Carrinho` são centralizadas no Builder

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

2.1 Conceitos Fundamentais

Stub: Um Test Double que retorna valores pré-configurados. Usado para **verificação de estado**, valida-se o resultado final do método testado.

Mock: Um Test Double que verifica **comportamento**, verifica-se se métodos foram chamados, quantas vezes, e com quais argumentos.

A diferença fundamental está no **foco da verificação**: - **Stub**: "O que o método retornou?" - **Mock**: "O método foi chamado corretamente?"

2.2 Identificação dos Test Doubles

*GatewayPagamento: **Stub***

O `GatewayPagamento` foi usado principalmente como **Stub** porque: - Retorna um valor pré-configurado: `{ success: true }` - É necessário para que o fluxo do teste continue (sem ele, o método retornaria `null`)

No entanto, também foi verificado seu **comportamento** (`toHaveBeenCalledWith`) para garantir que o desconto foi aplicado corretamente. Isso é uma verificação importante porque:
- Confirma que o valor passado para o gateway foi R$ 180,00 (desconto de 10% sobre R$ 200,00)
- Valida a regra de negócio de desconto para clientes Premium

Por que é principalmente um Stub? Porque o objetivo principal é fornecer uma resposta simulada para que o teste possa prosseguir. A verificação de comportamento é secundária, mas importante para validar a lógica de desconto.

*EmailService: **Mock***

O `EmailService` foi usado como **Mock** porque:
- O foco não é o valor de retorno (é `undefined`)
- O interesse está em **verificar se foi chamado** e **como foi chamado**
- Deseja-se garantir que o email foi enviado com os dados corretos após um pagamento bem-sucedido

Por que é um Mock? Porque o foco está em verificar o **comportamento** (chamadas do método), não o estado retornado. O email é um **efeito colateral** importante do processo de checkout que precisa ser validado.

*PedidoRepository: **Stub***

O `PedidoRepository` foi usado como **Stub** porque:
- Retorna um valor pré-configurado (o pedido salvo com ID)
- É necessário para que o fluxo continue (o email precisa do ID do pedido)
- Não verifica-se seu comportamento

Por que é um Stub? Porque o foco é apenas no **resultado** (pedido salvo), não em como ou quando foi chamado. O repositório é uma dependência necessária, mas não é o foco da verificação do teste.

---

3. Conclusão

O uso deliberado de **Padrões de Teste** (Builders e Test Doubles) é fundamental para criar uma suíte de testes sustentável e de alta qualidade. Através desta implementação, é possível observar os seguintes benefícios:

Prevenção de Test Smells

1. **Mystery Guest**: O Builder elimina valores “mágicos” e objetos criados inline, tornando os testes mais claros sobre quais dados estão sendo usados.

2. **Hard to Test**: Test Doubles isolam dependências externas, permitindo testar a lógica de negócio sem acoplamento a serviços reais.

3. **Test Code Duplication**: Builders centralizam a criação de objetos, eliminando código duplicado entre testes.

4. **Obscure Test**: A combinação de Builders expressivos e Test Doubles bem escolhidos torna os testes simples e fáceis de entender.

Contribuição para Sustentabilidade

Manutenibilidade:
- Mudanças na estrutura de domínio são isoladas nos Builders
- Test Doubles facilitam a refatoração ao reduzir acoplamento

Legibilidade:
- Testes focam no comportamento sendo testado, não em detalhes de setup
- Padrões consistentes facilitam a compreensão por novos membros da equipe

Confiabilidade:
- Test Doubles garantem que testes não falhem por problemas em dependências externas
- Builders garantem que objetos de teste estão sempre em estados válidos

Escalabilidade:
- Novos testes podem reutilizar Builders existentes
- Padrões estabelecidos facilitam a adição de novos cenários de teste

Em resumo, a aplicação consciente desses padrões transforma testes de código frágil e difícil de manter em uma suíte robusta que serve como documentação viva do sistema e proteção contra bugs/erros, contribuindo, significativamente, para a qualidade e sustentabilidade do software.