Pontifícia Universidade Católica de Minas Gerais

Engenharia de Software

Padrões de Teste em JavaScript

Object Mother, Data Builder e Test Doubles

Luiz Filipe Nery Costa

Matrícula: 805836

Testes de Software

Prof. Responsável

Belo Horizonte
2025

Sumário

1 Introdução

Este relatório apresenta a implementação de padrões de teste em um sistema de checkout de e-commerce. O trabalho teve como objetivo aplicar padrões reconhecidos pela literatura de engenharia de software para criar testes mais legíveis, manuteníveis e sustentáveis.

O sistema sob teste (SUT) consiste em um serviço de checkout que processa pedidos, aplicando regras de negócio como descontos para clientes premium, realizando cobranças via gateway de pagamento e enviando notificações por e-mail.

2 Padrões de Criação de Dados

2.1 Object Mother

O padrão Object Mother foi utilizado para a criação de objetos User, que são entidades simples e com variações limitadas. A implementação provê métodos estáticos que retornam instâncias pré-configuradas de usuários comuns e premium.

Listing 1: Implementação do UserMother

```
export class UserMother {
    static umUsuarioPadrao() {
        return new User(1, 'Joao Silva', 'joao@email.com', 'PADRAO
            ');
    }

    static umUsuarioPremium() {
        return new User(2, 'Maria Santos', 'premium@email.com', '
            PREMIUM');
    }
}
```

Este padrão é adequado para entidades que não precisam de muita customização entre os testes. No caso dos usuários, existem apenas dois tipos relevantes: padrão e premium.

2.2 Data Builder

O padrão Data Builder foi aplicado à classe Carrinho devido à sua complexidade e necessidade de flexibilidade. Um carrinho pode ter diferentes usuários, múltiplos itens ou estar vazio, o que tornaria inviável o uso de um Object Mother sem criar uma explosão de métodos.

Listing 2: Implementação do CarrinhoBuilder

```
export class CarrinhoBuilder {
    constructor() {
        this.user = UserMother.umUsuarioPadrao();
        this.itens = [new Item('Produto Padrao', 100)];
    }

    comUser(user) {
        this.user = user;
        return this;
```

```
    }

    comItens(itens) {
        this.itens = itens;
        return this;
    }

    vazio() {
        this.itens = [];
        return this;
    }

    build() {
        return new Carrinho(this.user, this.itens);
    }
}
```

2.3 Comparação: Antes vs. Depois

Antes (Setup Manual):

Listing 3: Setup manual complexo

```
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):

Listing 4: Setup usando Data Builder

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

2.4 Benefícios do Data Builder

- **Legibilidade:** A API fluente torna explícito apenas o que é relevante para cada cenário de teste.
- **Manutenibilidade:** Mudanças no construtor do Carrinho são isoladas no Builder, não afetando os testes.
- **Flexibilidade:** Permite criar carrinhos com configurações variadas sem duplicação de código.

- **Valores padrão:** O Builder fornece valores padrão sensatos, reduzindo o setup quando detalhes específicos não importam.

3 Padrões de Test Doubles

3.1 Stubs: Verificação de Estado

No teste de falha de pagamento, utilizamos stubs para controlar o comportamento das dependências externas. O GatewayPagamento foi configurado como stub para retornar uma resposta de falha:

Listing 5: Uso de Stub no teste de falha

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

O stub controla o fluxo do teste, mas não verificamos se ele foi chamado. O foco está no estado resultante: esperamos que o método `processarPedido` retorne `null`.

Listing 6: Verificação de Estado

```
const pedido = await checkoutService.processarPedido(carrinho,
   '1234-5678');
expect(pedido).toBeNull();
```

3.2 Mocks: Verificação de Comportamento

No teste de sucesso com cliente Premium, utilizamos tanto stubs quanto mocks. O GatewayPagamento e o PedidoRepository atuam como stubs, controlando o fluxo, mas o EmailService é um mock, pois verificamos se ele foi chamado corretamente:

Listing 7: Uso de Mock para EmailService

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

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

3.3 Diferença: Stub vs. Mock

GatewayPagamento (Stub):

- Objetivo: Controlar o retorno (sucesso ou falha) para direcionar o fluxo do teste.
- Verificação: Focada no estado resultante (valor cobrado, pedido criado).

- Justificativa: Interessa o que o sistema faz com a resposta, não quantas vezes chamou.

EmailService (Mock):

- Objetivo: Verificar se o sistema notifica o cliente corretamente.
- Verificação: Focada no comportamento (chamada realizada, argumentos corretos).
- Justificativa: Enviar e-mail é um efeito colateral crítico que deve ser testado.

4 Implementação dos Testes

4.1 Teste de Falha no Pagamento

Este teste valida o cenário onde o gateway de pagamento rejeita a cobrança. Utilizamos stubs para todas as dependências e verificamos que o sistema retorna `null`, não salvando o pedido nem enviando e-mail.

Listing 8: Teste de falha no pagamento

```
describe('quando o pagamento falha', () => {
    it('deve retornar null', async () => {
        const carrinho = new CarrinhoBuilder()
            .comItens([new Item('Notebook', 3000)])
            .build();

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

        const checkoutService = new CheckoutService(
            gatewayStub,
            repositoryDummy,
            emailDummy
        );

        const pedido = await checkoutService.processarPedido(
            carrinho,
            '1234-5678'
        );

        expect(pedido).toBeNull();
    });
});
```

4.2 Teste de Cliente Premium

Este teste valida o cenário completo de sucesso para cliente premium, verificando o desconto aplicado e o envio do e-mail de confirmação.

Listing 9: Teste de cliente premium

```
describe('quando um cliente Premium finaliza a compra', () => {
    it('deve aplicar desconto de 10% e enviar email', async () => {
        const usuarioPremium = UserMother.umUsuarioPremium();
        const carrinho = new CarrinhoBuilder()
            .comUser(usuarioPremium)
            .comItens([
                new Item('Mouse', 100),
                new Item('Teclado', 100)
            ])
            .build();

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

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

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

        await checkoutService.processarPedido(carrinho,
           '1234-5678');

        expect(gatewayStub.cobrar).toHaveBeenCalledWith(180,
           '1234-5678');
        expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
        expect(emailMock.enviarEmail).toHaveBeenCalledWith(
            'premium@email.com',
            'Seu Pedido foi Aprovado!',
            'Pedido PED-001 no valor de R$180'
        );
    });
});
```

5 Prevenção de Test Smells

Os padrões implementados ajudam a evitar diversos test smells:

5.1 Setup Obscuro

O CarrinhoBuilder elimina o setup obscuro ao tornar explícito apenas o que é relevante para cada teste. Em vez de passar múltiplos parâmetros ao construtor, usamos métodos fluentes que documentam a intenção.

5.2 Dependência de Dados Mágicos

O UserMother centraliza a criação de usuários de teste, evitando que strings mágicas e IDs arbitrários sejam espalhados pelos testes.

5.3 Testes Frágeis

O uso de Builders isola os testes de mudanças na implementação das classes de domínio. Se o construtor do Carrinho mudar, apenas o Builder precisa ser atualizado.

5.4 Verificação Excessiva

Ao diferenciar claramente entre Stubs (controle de fluxo) e Mocks (verificação de comportamento), evitamos verificar interações que não são relevantes para cada cenário de teste.

6 Conclusão

A aplicação deliberada de padrões de teste demonstrou benefícios concretos na qualidade da suíte de testes. O uso do Data Builder para objetos complexos como Carrinho evitou a explosão de métodos que ocorreria com um Object Mother, enquanto manteve a legibilidade e facilitou a manutenção.

A distinção entre Stubs e Mocks permitiu criar testes focados: Stubs controlam o fluxo baseando-se em verificação de estado, enquanto Mocks validam comportamentos críticos como o envio de notificações. Esta abordagem resulta em testes menos frágeis e mais expressivos.

Os padrões implementados contribuem para uma suíte de testes sustentável ao:

- Reduzir duplicação através de builders e factories centralizados
- Aumentar a legibilidade com APIs fluentes e nomes expressivos
- Facilitar a manutenção isolando mudanças em poucos pontos
- Prevenir test smells através de práticas estabelecidas pela literatura

Estes benefícios tornam a suíte de testes mais confiável e menos propensa a falsos positivos, garantindo que ela continue sendo um ativo valioso ao longo da evolução do sistema.

7 Referências

- FOWLER, Martin. Mocks Aren't Stubs. Disponível em: `https://martinfowler.com/articles/mocksArentStubs.html`. Acesso em: 09 nov. 2025.
- MESZAROS, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.
- JEST. Mock Functions. Disponível em: `https://jestjs.io/docs/mock-functions`. Acesso em: 09 nov. 2025.

- FREEMAN, Steve; PRYCE, Nat. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009.