Implementando Padrões de Teste

Aluno: Victor Reis Carlota
Disciplina: Testes de Software
Trabalho: Implementando Padrões de Teste (Test Patterns)

---

1. Introdução

Neste trabalho, o objetivo foi aplicar **padrões de teste** para prevenir *test smells* e construir uma suíte de testes mais legível, sustentável e confiável.
O sistema em questão simula um serviço de **Checkout de E-commerce**, responsável por processar pedidos, efetuar cobranças e enviar confirmações por e-mail.

O foco do exercício foi utilizar padrões de **criação de dados de teste** (*Object Mother* e *Data Builder*) e padrões de **dublês de teste** (*Stubs* e *Mocks*), além de seguir o modelo *AAA (Arrange, Act, Assert)* para estruturar testes claros e bem organizados.

---

2. Padrões de Criação de Dados (Builders)

2.1. Object Mother

O padrão **Object Mother** foi utilizado para criar objetos simples e fixos, como o usuário.
A classe `UserMother` encapsula a criação de usuários com perfis diferentes, como o usuário padrão e o premium.

Exemplo:

```
export class UserMother {
  static umUsuarioPadrao() {
    return new User(1, "Cliente Comum", "padrao@email.com", "PADRAO");
  }

  static umUsuarioPremium() {
    return new User(2, "Cliente Premium", "premium@email.com", "PREMIUM");
  }
}
```

Isso evita duplicação de código e melhora a clareza dos testes, já que o foco passa a ser o **cenário de teste** e não a criação de objetos.

2.2. Data Builder

O padrão **Data Builder** foi aplicado à entidade `Carrinho`, um objeto mais complexo que pode variar em número de itens e tipos de usuário. Enquanto o *Object Mother* seria inflexível (exigindo múltiplos métodos), o *Builder* permite construir variações fluentes e legíveis.

Antes (sem Builder)

```
const carrinho = new Carrinho(
  [new Item("Produto 1", 200)],
  new User(1, "Cliente Premium", "premium@email.com", "PREMIUM")
);
```

Depois (com Builder)

```
const carrinho = new CarrinhoBuilder()
  .comUser(UserMother.umUsuarioPremium())
  .comItens([new Item("Produto 1", 200)])
  .build();
```

A API fluente do Builder deixa o *setup* explícito e fácil de entender. Além disso, o padrão ajuda a eliminar o *test smell* **Obscure Setup**, pois deixa claro quais partes do objeto são relevantes para o teste.

---

3. Padrões de Test Doubles (Stubs e Mocks)

Para isolar dependências externas e focar apenas na lógica do `CheckoutService`, foram usados **dublês de teste** com funções do Jest (`jest.fn()`).

3.1. Stub (Verificação de Estado)

O **Stub** foi usado para simular respostas controladas de serviços externos. No teste de falha no pagamento, o `GatewayPagamento` foi implementado como um *Stub*, retornando `{ success: false }`. Isso permite validar o *estado final* do método sem depender de chamadas reais.

Exemplo:

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

Verificação:

```
expect(resultado).toBeNull();
```

3.2. Mock (Verificação de Comportamento)

O **Mock** foi utilizado para validar o *comportamento esperado* — se métodos foram chamados e com quais argumentos. No teste de sucesso, o `EmailService` foi um *Mock*, permitindo garantir que a confirmação foi enviada corretamente.

Exemplo:

```
expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
expect(emailMock.enviarEmail).toHaveBeenCalledWith(
  "premium@email.com",
  "Seu Pedido foi Aprovado!",
  expect.anything()
);
```

Assim, os *Mocks* focam na **verificação de comportamento**, enquanto os *Stubs* lidam com a **verificação de estado**.

---

4. Estrutura dos Testes (AAA Pattern)

Todos os testes seguiram o padrão **Arrange, Act, Assert**, que divide a lógica em três seções claras:

1. **Arrange:** Preparar os dados, dependências e objetos necessários.
2. **Act:** Executar a ação sob teste.
3. **Assert:** Verificar o resultado ou comportamento esperado.

Exemplo completo (cenário Premium):

```
// Arrange
const userPremium = UserMother.umUsuarioPremium();
const carrinho = new CarrinhoBuilder().comUser(userPremium).comItens([new Item("Produto 1",

const gatewayStub = { cobrar: jest.fn().mockResolvedValue({ success: true }) };
const pedidoRepoStub = { salvar: jest.fn().mockResolvedValue({ id: 1 }) };
const emailMock = { enviarEmail: jest.fn() };

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

// Act
await checkoutService.processarPedido(carrinho, "cartao123");

// Assert
expect(gatewayStub.cobrar).toHaveBeenCalledWith(180, expect.anything());
expect(emailMock.enviarEmail).toHaveBeenCalledTimes(1);
```

---

5. Conclusão

O uso dos padrões **Object Mother**, **Data Builder**, **Stub** e **Mock** resultou em uma suíte de testes **limpa, legível e fácil de manter**. Esses padrões permitem separar a criação de dados de teste da lógica de verificação, reduzir duplicação e isolar dependências externas.

Com isso, o processo de desenvolvimento se torna mais ágil e confiável, prevenindo *Test Smells* como *Obscure Setup*, *Fragile Tests* e *Test Duplication*. Além disso, o uso do padrão AAA torna os testes autoexplicativos e consistentes, promovendo qualidade e sustentabilidade no ciclo de vida do software.

Repositório: https://github.com/victor-reis-carlota/test-pattern

“‘