Relatório — Padrões de Teste (Test Patterns)

Disciplina: Testes de Software

Trabalho: Implementando Padrões de Teste

Aluno: Gabriel Faria de Oliveira

Matrícula: 805560

---

1. Introdução

Este relatório descreve a implementação dos padrões de criação de dados (Object Mother e Data Builder) e dos Test Doubles (Stubs e Mocks) aplicados à suíte de testes do serviço de Checkout de um e-commerce. O objetivo foi construir testes legíveis, isolados e confiáveis, prevenindo Test Smells como Setup Obscuro e Testes Frágeis.

2. Padrões de Criação de Dados (Builders)

Por que usar um `CarrinhoBuilder` em vez de um `CarrinhoMother`?

O `User` é uma entidade simples e com valores relativamente estáveis entre testes (tipos PADRAO/PREMIUM). Para isso, um Object Mother (`UserMother`) fornece métodos fixos e legíveis: `umUsuarioPadrao()` e `umUsuarioPremium()` .

Já `Carrinho` é um objeto composto: pode ter diferentes usuários, vários itens, preços variados ou estar vazio. Um `CarrinhoMother` exigiria uma explosão de métodos (`carrinhoComUmItem` , `carrinhoComDoisItens` , `carrinhoPremiumComDesconto` , etc.), o que leva novamente ao Test Smell "Setup Obscuro" quando muitos métodos escondem detalhes importantes.

O `CarrinhoBuilder` resolve isso oferecendo uma API fluente e configurável:

- Construtor com valores sensatos por padrão (ex.: 1 item de R$100).
- Métodos fluentes para alterar apenas o que importa: `.comUser(user)` , `.comItens(itens)` , `.vazio()` .
- Método `.build()` que retorna a instância final do `Carrinho` .

Essa abordagem torna explícito no teste apenas o que o cenário precisa, reduzindo duplicação e melhorando manutenção.

Exemplo — Teste "Antes" e "Depois"

Antes (setup manual complexo — exemplo hipotético):

```
// Antes: código de setup verboso embutido em cada teste
const user = new User(2, 'Maria Premium', 'premium@email.com', 'PREMIUM');
const itens = [ new Item('Produto A', 150), new Item('Produto B', 50) ];
const carrinho = new Carrinho(user, itens);
// ...mais linhas para configurar mocks/stubs...
```

Depois (usando `CarrinhoBuilder` — mais legível):

```
const userPremium = UserMother.umUsuarioPremium();
const carrinho = new CarrinhoBuilder()
  .comUser(userPremium)
  .comItens([{ nome: 'Produto Caro', preco: 200 }])
  .build();
```

Comparação e justificativa:

- O builder reduz ruido: o leitor do teste vê em poucas linhas que o carrinho contém um item de R$200 pertencente a um usuário premium.
- Se um teste precisar de variações, ajustar no builder (`.vazio()` ou `.comItens(...)`) é trivial e não polui o teste com detalhes irrelevantes.
- Em mudanças de domínio (ex.: adicionar campo `categoria` a `Item`), centralizar a criação no builder diminui pontos de modificação.

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

Teste escolhido: "sucesso Premium" (compra de cliente Premium)

Resumo do cenário do teste:

- Usuário Premium realiza compra com itens totalizando R$200.
- Aplicamos desconto de 10% para clientes Premium (total a cobrar = R$180).
- `GatewayPagamento` deve receber o valor correto e retornar sucesso.
- `PedidoRepository` salva e retorna o pedido com `id` .
- `EmailService` deve ser chamado para notificar o usuário.

Dependências usadas no teste:

- `GatewayPagamento` — Stub
- `PedidoRepository` — Stub
- `EmailService` — Mock

Por que `GatewayPagamento` é um Stub?

- Objetivo no teste: controlar o fluxo do SUT (System Under Test) — simular sucesso do pagamento sem testar o gateway real.
- Usamos `gatewayStub.cobrar = jest.fn().mockResolvedValue({ success: true })`.
- Validamos a *estado/resultado* do fluxo (o pedido final retornado e o valor cobrado) e não o comportamento interno do gateway.
- Em testes de verificação de estado (state verification), stubs são ideais: fornecem respostas determinísticas para o SUT.

Por que `EmailService` é um Mock?

- Além de controlar retorno, queremos afirmar que uma interação ocorreu: que o serviço de e-mail foi chamado exatamente uma vez com argumentos específicos (destinatário, assunto, corpo).
- Usamos `emailMock.enviarEmail = jest.fn()` e assertions como `toHaveBeenCalledTimes(1)` e `toHaveBeenCalledWith(...)`.
- Esse é um exemplo clássico de verificação de comportamento (behavior verification): o teste garante que a função externa foi invocada corretamente como efeito colateral do fluxo.

Observação sobre `PedidoRepository`:

- Foi usado como stub que retorna o `pedidoSalvo` (com `id`), porque aqui nos interessa o valor retornado para embutir no corpo do e-mail e no retorno do método `processarPedido`.

Resumo: Estado x Comportamento

- Stubs: controlam retornos e permitem testar o comportamento do SUT em diferentes cenários (ex.: pagamento falha/age com sucesso). Foco em State Verification.
- Mocks: verificam interações (quantas vezes, com quais argumentos). Foco em Behavior Verification.

4. Conclusão

A aplicação deliberada de Builders (para criação de dados) e Test Doubles (stubs e mocks) melhora significativamente a qualidade da suíte de testes. Benefícios observados:

- Legibilidade: testes ficam curtos e focados. O `CarrinhoBuilder` deixa claro o que muda entre cenários.
- Manutenção: mudanças no formato de entidades exigem alteração em um ponto (builder) e não em muitos testes espalhados.
- Robustez e isolamento: usando stubs para controlar dependências externas, os testes ficam determinísticos e rápidos. Mocks permitem garantir efeitos colaterais importantes (envio de e-mail) sem depender de serviços reais.
- Prevenção de Test Smells: essas práticas reduzem Setup Obscuro (setup escondido em muitos lugares), Testes Frágeis (dependência de serviços externos) e duplicação.

Em suma, o uso combinado de Data Builders e Test Doubles resulta em uma suíte de testes mais sustentável, que favorece refatoração segura e feedback rápido no desenvolvimento.