Relatório de Implementação de Padrões de Teste

Capa

Disciplina: Teste de Software **Nome do Trabalho:** Test Patterns **Nome Completo:** Henrique Jardim Melo

---

Padrões de Criação de Dados (Builders)

Object Mother vs. Data Builder

O padrão **Object Mother** (`UserMother.js`) foi utilizado para a entidade `User` (Usuário). Este padrão é ideal para a criação de objetos simples e com estado fixo, como um "Usuário Padrão" ou um "Usuário Premium". A principal vantagem é a simplicidade e a garantia de que o objeto retornado é sempre o mesmo, reduzindo a duplicação de código de setup.

Por outro lado, o padrão **Data Builder** (`CarrinhoBuilder.js`) foi escolhido para a entidade `Carrinho` (Carrinho de Compras). O carrinho é um objeto **complexo** e **volátil**, pois pode ter diferentes usuários, diferentes quantidades de itens, ou estar completamente vazio.

Por que o CarrinhoBuilder foi usado em vez de um CarrinhoMother?

Um `CarrinhoMother` exigiria a criação de um método estático para cada variação possível do carrinho (ex: `CarrinhoMother.comUmItem()`, `CarrinhoMother.comDezItens()`, `CarrinhoMother.comUsuarioPremium()`, etc.). Isso levaria a uma **explosão de métodos** e ao Test Smell de **Setup Obscuro** (Obscure Setup), onde o setup do teste se torna longo e difícil de entender.

O `CarrinhoBuilder` resolve este problema ao oferecer uma **API fluente** que permite a customização sob demanda, expondo apenas as variações relevantes para o teste em questão.

Exemplo de Teste: "Antes" e "Depois"

Cenário	Setup “Antes” (Manual e Complexo)	Setup “Depois” (Usando Data Builder)
Carrinho com 2 itens e usuário Premium	`javascript\nconst user = new User(2, 'P', 'p@e.com', 'PREMIUM');\nconst item1 = new Item('I1', 100.00);\nconst item2 = new Item('I2', 100.00);\nconst carrinho = new Carrinho(user, [item1, item2]);`	`javascript\nconst user = UserMother.umUsuarioPremium();\nconst carrinho = CarrinhoBuilder.umCarrinho()\n .comUser(user)\n .comItens([new Item('I1', 100), new Item('I2', 100)])\n .build();`

Justificativa de Legibilidade e Manutenção

O Builder melhora a legibilidade e manutenção do teste porque: 1. **Clareza:** A API fluente (`.comUser()`, `.comItens()`) torna o setup **explícito** e **autodocumentado**. O leitor do teste entende imediatamente quais características do objeto são importantes para o cenário. 2. **Foco:** Evita a necessidade de criar objetos com dezenas de linhas de código de setup irrelevante para o teste. O teste foca apenas na customização necessária, combatendo o Test Smell de **Setup Obscuro**. 3. **Manutenção:** Se a classe `Carrinho` mudar (ex: adicionar um novo campo obrigatório), apenas o `CarrinhoBuilder` precisa ser atualizado, e não todos os testes que o utilizam.

Padrões de Test Doubles (Mocks vs. Stubs)

Análise do Teste de Sucesso Premium (Etapa 5)

O teste `quando um cliente Premium finaliza a compra` verifica o fluxo completo de sucesso, que envolve três dependências externas: `GatewayPagamento`, `PedidoRepository` e `EmailService`.

Dependência	Padrão Utilizado	Justificativa
`GatewayPagamento`	**Stub**	O `CheckoutService` precisa da **resposta** do Gateway (se o pagamento foi `success: true`) para decidir se continua o fluxo. O Stub fornece essa resposta pré-definida, permitindo que o teste se concentre na **Verificação de Estado** (State Verification) do SUT (o pedido deve ser salvo e retornado).
`PedidoRepository`	**Stub**	O SUT precisa que o repositório retorne o pedido salvo (com ID) para poder enviá-lo no e-mail. O Stub fornece o objeto `pedidoSalvo`, permitindo que o teste continue. O foco é no **Estado** do SUT.

`EmailService`	**Mock**	O SUT não depende do **retorno** do `EmailService` para continuar (o envio de e-mail é um efeito colateral). O teste precisa garantir que o `EmailService` foi **chamado** exatamente uma vez, com os **argumentos corretos** (e-mail do cliente Premium e valor com desconto). O Mock é usado para **Verificação de Comportamento** (Behavior Verification), garantindo que o efeito colateral esperado ocorreu.

Explicação: Stub vs. Mock

A distinção entre Stub e Mock, popularizada por Martin Fowler, reside no seu papel na verificação do teste:

- **Stub (Verificação de Estado):** É um objeto que fornece respostas pré-programadas para chamadas feitas durante o teste. Ele é usado para controlar o fluxo de execução do SUT. O teste verifica o **estado** final do SUT (ex: o método `processarPedido` retornou o objeto `pedidoSalvo` ou `null`). O `GatewayPagamento` e o `PedidoRepository` são Stubs porque o SUT precisa de seus retornos para prosseguir.
- **Mock (Verificação de Comportamento):** É um objeto que, além de fornecer respostas (como um Stub), tem expectativas pré-programadas sobre as chamadas que deve receber. O teste verifica se o SUT interagiu com o Mock da maneira esperada. O `EmailService` é um Mock porque o teste precisa garantir que o SUT se **comportou** corretamente ao chamar o serviço de e-mail com o valor já com o desconto aplicado.

Conclusão

O uso deliberado de Padrões de Teste, como **Object Mother**, **Data Builder**, **Stubs** e **Mocks**, é fundamental para a construção de uma suíte de testes **sustentável** e **eficaz**.

Os **Builders** combatem o Test Smell de **Setup Obscuro**, tornando a criação de dados de teste complexos mais legível e flexível. Ao isolar a lógica de criação, eles também garantem que a manutenção do setup seja centralizada.

Os **Test Doubles** (Stubs e Mocks) combatem o Test Smell de **Testes Frágeis** ao isolar o SUT de dependências externas voláteis. A distinção clara entre Stubs (para controlar o fluxo e verificar o estado) e Mocks (para verificar o comportamento e efeitos colaterais) garante que cada teste seja focado, seguindo o princípio de **Single Responsibility Principle** para testes.

Ao aplicar esses padrões, transformamos testes que seriam longos, frágeis e difíceis de manter em testes **limpos**, **isolados** e **auto-documentados**, contribuindo diretamente para a qualidade e a confiança no código de produção.