Implementando Padrões de Teste (Test Patterns)

Autor: João Francisco Carvalho Soares de Oliveira Queiroga

Disciplina: Teste de Software

Projeto: Implementando Padrões de Teste (Test Patterns)

Padrões de Criação de Dados (*Builders*)

O padrão *Data Builder* foi adotado para facilitar a criação de dados complexos nos testes da aplicação, especialmente para a entidade `Carrinho`. O `CarrinhoBuilder` foi escolhido em vez de um *CarrinhoMother* por permitir maior flexibilidade e clareza na configuração dos objetos de teste.

Um *Object Mother* normalmente cria objetos estáticos e pouco configuráveis. Já o *Builder* possibilita construir objetos passo a passo, com métodos encadeáveis, permitindo testar diferentes cenários sem duplicar código.

Antes (setup manual complexo):

```
const user = new User(1, "Usuário Premium", "premium@exemplo.com", "PREMIUM");
const itens = [
  new Item("Livro", 50),
  new Item("Teclado", 150)
];
const carrinho = new Carrinho(user, itens);
```

Depois (usando o CarrinhoBuilder):

```
const carrinho = new CarrinhoBuilder()
  .comUsuarioPremium()
  .comItensPadrao()
  .build();
```

Com o *Builder*, o teste torna-se mais legível e expressivo, deixando claro o cenário em teste. Além disso, as mudanças na estrutura das classes não exigem grandes alterações nos testes, apenas no *Builder*, centralizando a manutenção.

Padrões de *Test Doubles* (*Mocks* e *Stubs*)

No teste de “sucesso Premium” (Etapa 5), foram aplicados dois tipos distintos de *test doubles*:

- **Stub:** `GatewayPagamento`
- **Mock:** `EmailService`

O `GatewayPagamento` foi usado como *Stub* porque seu papel é apenas retornar um valor pré-definido, simulando o comportamento de um pagamento bem-sucedido. O foco é verificar o **estado final** do sistema após a operação, e não se o método foi chamado corretamente.

Por outro lado, o `EmailService` foi utilizado como *Mock* porque é necessário verificar o **comportamento** — ou seja, se o serviço foi chamado com os parâmetros corretos após a finalização da compra. Isso garante que o sistema não apenas processe o pagamento, mas também execute ações esperadas de comunicação.

Conclusão

O uso deliberado de *Data Builders* e *Test Doubles* reduz a complexidade e previne *Test Smells*, como setups redundantes e testes frágeis. O *CarrinhoBuilder* melhorou a clareza e a reutilização dos testes, enquanto o uso de *Stubs* e *Mocks* permitiu isolar dependências externas, focando apenas no comportamento da unidade em teste. Essas práticas contribuem para uma suíte de testes sustentável, de fácil manutenção e resistente a mudanças na base de código.