Maria Eduarda Gonçalves de Souza Ferreira

1. Introdução

Este trabalho teve como objetivo aplicar alguns dos principais padrões de teste de software, mostrando na prática como eles ajudam a deixar os testes mais claros, organizados e fáceis de manter. Os padrões escolhidos foram Object Mother, Data Builder, Stub e Mock, usados em uma simulação de testes para o módulo de checkout de um e-commerce. A ideia foi resolver problemas comuns de testes mal estruturados, como setups longos, código repetido e dependência de recursos externos (como banco de dados ou serviços reais). Com esses padrões, os testes ficam mais previsíveis, expressivos e independentes, o que melhora muito a qualidade do projeto.

2. Object Mother e Data Builder

O Object Mother é usado quando precisamos criar instâncias padrão de classes usadas várias vezes, de forma centralizada. No meu caso, criei a classe UserMother, que gera dois tipos de usuário: um padrão e outro premium. Assim, qualquer teste que precisar de um usuário pode simplesmente chamar UserMother.umUsuarioPremium(), por exemplo.

Já o Data Builder é mais indicado quando o objeto tem muitas variações possíveis. O CarrinhoBuilder foi criado para montar carrinhos de compra com uma API fluente, podendo alterar apenas o que o teste realmente precisa. Isso evita a criação de dezenas de métodos fixos e deixa o código bem mais limpo.

Exemplo:

Antes (sem Builder):

```
const user = new User({ id: "U1", nome: "Fulano", tipo: "REGULAR" });
const itens = [ new Item("SKU-1", "Mouse", 100, 1) ];
const carrinho = new Carrinho(user, itens);
```

Depois (com Builder):

```
const carrinho = new CarrinhoBuilder()
  .comUser(UserMother.umUsuarioPremium())
  .comItens([ new Item("SKU-1", "Mouse", 100, 1) ])
  .build();
```

Além de mais legível, o setup fica curto e direto. Isso ajuda muito quando há vários testes semelhantes.

3. Stubs, Mocks e Test Doubles

Nos testes do CheckoutService, usei stubs e mocks para substituir dependências externas, como o gateway de pagamento, o repositório de pedidos e o serviço de e-mail. A ideia é que o teste não dependa de sistemas reais e que eu possa controlar exatamente o comportamento dessas dependências.

No cenário de falha no pagamento, o stub do gateway retorna { success: false }. Com isso, verifico se o método processarPedido() retorna null e se nenhuma outra ação é feita (não salva pedido nem envia e-mail).

No cenário de sucesso com cliente premium, o stub do gateway retorna { success: true }. O serviço aplica 10% de desconto, salva o pedido e envia um e-mail de confirmação. Nesse caso, o Mock é usado para confirmar se o e-mail foi realmente enviado, para quem e quantas vezes.

Essas práticas deixam os testes mais focados no comportamento da aplicação e evitam dependência de integrações externas, que poderiam falhar ou gerar resultados diferentes.

4. Estrutura AAA

Todos os testes foram escritos seguindo o padrão AAA (Arrange, Act, Assert):

1. Arrange: monta o cenário e as dependências (usando os Builders e Mothers).
2. Act: executa o método principal, no caso, processarPedido().
3. Assert: verifica o resultado (estado ou comportamento esperado).

Essa estrutura ajuda a manter todos os testes com o mesmo formato e facilita entender o que cada parte faz, mesmo para quem nunca viu o código antes.

5. Resultados e Aprendizados

A implementação desses padrões trouxe vários ganhos práticos. Os testes ficaram mais claros e legíveis, com menos código repetido e cenários mais fáceis de entender. Além disso, ao usar stubs e mocks, os testes se tornaram totalmente independentes, permitindo rodá-los sem precisar de banco ou APIs externas. Também percebi como o uso de padrões de teste faz a gente pensar melhor na arquitetura do código, já que o design precisa ser mais modular e com dependências injetáveis. No final, isso melhora não só os testes, mas o projeto como um todo.

6. Conclusão

De forma geral, os padrões de teste aplicados neste trabalho mostraram como pequenas mudanças na forma de escrever testes podem gerar grandes benefícios em clareza, manutenção e confiabilidade.

O uso de Object Mother e Data Builder simplificou o preparo de dados, enquanto os Stubs e Mocks garantiram isolamento e controle total sobre os cenários de teste. Com isso, o código de teste ficou mais limpo e fácil de entender, além de seguir boas práticas profissionais que são muito valorizadas no mercado.

Como próximos passos, seria interessante criar mais casos cobrindo situações como carrinho vazio, erro no envio de e-mail ou falha no repositório. Também seria bom medir a cobertura de testes usando ferramentas do Jest para avaliar o quanto do sistema está realmente sendo testado.