Testes automatizados

12/08/2025

Na Aula de hoje

- identificar como verificar a funcionalidade correta do código
- moldar seu código para ser mais testável
- pesquisar teste caixa branca, teste caixa preta, teste dinâmico versus teste estático.

Particionamento de Equivalência

Técnica de teste de software (caixa-preta) que divide entradas ou saídas em **classes de equivalência** — grupos onde os valores são considerados equivalentes para fins de teste.

Permite testar um número **reduzido de casos**, sem perder a efetividade.

Por que usar essa técnica?

- Reduz o número de testes necessários.
- Garante cobertura significativa com menos esforço.
- Foca em valores representativos.
- Muito usada em sistemas com entradas numéricas ou categóricas.

Classe de Equivalência

- Um **intervalo de valores** ou conjunto de dados com o mesmo comportamento esperado.
- Um teste dentro da classe é **suficiente** para representar todos os outros valores.

Se um valor da classe falha, os demais também têm chance de falhar.

Ex: contratação por idade

Faixa etária	Resposta do sistema
0–16	NÃO
16–18	PARCIAL
18–55	INTEGRAL
55–99	NÃO

Casos de teste possíveis:

- 10 anos → NÃO
- 17 anos → PARCIAL
- 30 anos → INTEGRAL
- 60 anos → NÃO

Como aplicar o Particionamento

1. **Identifique as classes de equivalência** (válidas e inválidas).
2. Crie **um teste para cada classe válida**.
3. Crie **um teste para cada classe inválida**.
4. Adicione testes extras se houver tempo ou risco alto.

Tipos comuns de classes

Dados contínuos (ex: renda)

- Válido: `R$1.000` a `R$83.000`
- Inválidos: abaixo de `R$1.000` ou acima de `R$83.000`

Dados discretos (ex: 1 a 5 imóveis)

- Válido: 3 imóveis
- Inválido: -2, 8

Texto (ex: identificadores válidos)

- Válido: `abc12`
- Inválido: `1nome`, `cont*1`

Aplicabilidade

✅ Vantagens:

- Reduz testes repetitivos
- Efetivo para entradas com intervalos e condições
- Funciona para testes de unidade, integração, sistema e aceitação

❌ Limitações:

- Depende de requisitos claros
- Pode não detectar erros "escondidos" ou exceções específicas no código

---

Conclusão

Particionamento de Equivalência é uma técnica **eficiente, simples e poderosa** para reduzir o esforço de testes sem sacrificar qualidade.

Ideal para validar funcionalidades com base em entradas organizadas por intervalos ou regras claras.

Como testar?

Testar não é só executar o código: é **garantir que ele funcione corretamente em diferentes situações**. Para isso, é preciso **planejar, estruturar e moldar o código** pensando em testes desde o início.

Por que pensar em "como testar"?

- Testes bem-feitos previnem bugs antes de irem para produção.
- Saber **como testar** leva a um código **mais limpo, coeso e confiável**.
- Quanto mais cedo você pensar nos testes, **menos retrabalho terá depois**.

Como verificar a funcionalidade

Antes de escrever testes, pergunte:

- O que essa função/componente deveria fazer?
- Como saberei que ela funcionou corretamente?
- Existe **entrada e saída claramente definidas**?
- Há efeitos colaterais (como escrever em arquivos, banco de dados, etc.)?

Use isso para definir:

- Quais são os **valores esperados**?
- Quais são os **casos especiais** ou limites?

Exemplo de função

```
def somar(a, b):
    return a + b
```

Como testar?

- Entrada: `2, 3` → Saída esperada: `5`
- Entrada: `-1, 1` → Saída esperada: `0`
- Entrada: `0, 0` → Saída esperada: `0`

Ex: comportamento inesperado

```
def dividir(a, b):
    return a / b
```

Como testar?

- Entrada: `6, 2` → Saída esperada: `3`
- Entrada: `5, 0` → Espera-se erro (divisão por zero)

→ **É importante testar exceções e falhas também!**

Moldando seu código para ser testável

Um código testável é aquele que:

- É **modular** (pequenas funções independentes)
- **Evita efeitos colaterais** (como acessar disco ou rede sem necessidade)
- Usa **valores de entrada/saída claros**
- Permite **substituir dependências** por simulações (mocks)

Como tornar o código mais testável

- Separe **lógica de negócios** da interface (ex: do terminal ou do navegador).
- Evite variáveis globais ou estados compartilhados.
- Prefira **funções puras** sempre que possível.
- Use **injeção de dependência** para facilitar testes isolados.
- Escreva funções pequenas e com responsabilidade única.

Antes de escrever o teste, reflita:

- O que pode dar errado?
- O que **não pode acontecer**?
- Quais são as **entradas válidas, inválidas e limites**?

Conclusão

Saber **como testar** é mais do que saber usar uma ferramenta. É **pensar como desenvolvedor e testador ao mesmo tempo**, antecipando falhas e preparando o código para ser validado com segurança.

→ Bons testes dependem de **bom código**. E bom código é, quase sempre, **mais fácil de testar**.

Atividade

- Pesquisem sobre teste caixa branca, teste caixa preta, teste dinâmico versus teste estático.
- Quais casos de testes podemos ter pra esse código? Façam a tabela de entrada, e saída esperada. (Particionamento de equivalência)

```
function situacaoAluno(nota: number){
    if(nota <= 4){
        return "Reprovado sem direito a exame"
    }
    if(nota > 4 &&  nota < 6){
        return "Reprovado com direito a exame"
    }
    if(nota == 10){
        return "Aprovado e condecorado"
    }
    return "Aprovado"
}
```

---

classe	Saida
0 - 4	"Reprovado sem direito a exame"
5.0 - 5.9	"Reprovado com direito a exame"
6 - 9.9	"Aprovado"
10	"Aprovado e condecorado"