

IF BOM É

IF ~~MORTO~~



THULIO BITTENCOURT

ENTERRANDO O 'IF' COM ESTILO: SEGREDOS PARA UMA
PROGRAMAÇÃO **DELPHI** SEM MORTOS-VIVOS
CONDICIONAIS



Thulio Bittencourt

Embarcadero MVP desde 2017
Founder Academia do Código



@thuliobittencourt



**Anti-IF
Programming**

CEO e fundador da Combinant Dynamics, um centro de excelência em desenvolvimento de software com sede em Dubai que trabalha com muitas das maiores empresas do mundo.

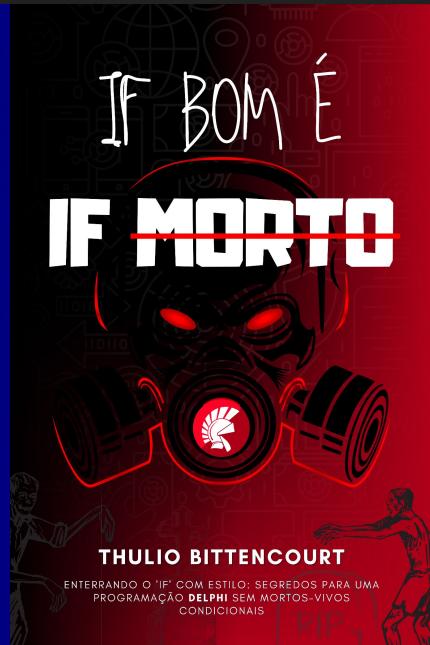
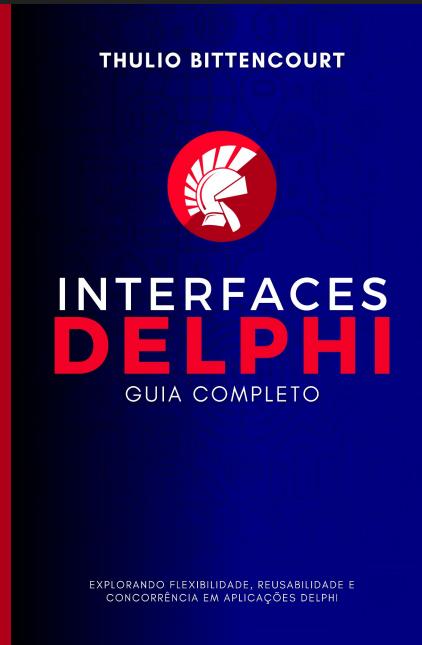
Ele criou a Técnica Pomodoro®, uma renomada ferramenta de gerenciamento de tempo usada por milhões de pessoas em todo o mundo, enquanto um estudante universitário procurava uma maneira de fazer mais em menos tempo. O foco principal de Francesco sempre foi melhorar a produtividade e a eficiência, encontrando maneiras de atingir melhores resultados com menos tempo e menos esforço.

Francesco trabalha na vanguarda da indústria de software há mais de 20 anos. Em uma carreira que abrange startups, multinacionais e consultoria freelance, ele orientou milhares de profissionais, desenvolvedores, gerentes e equipes de software.

antiifprogramming.com

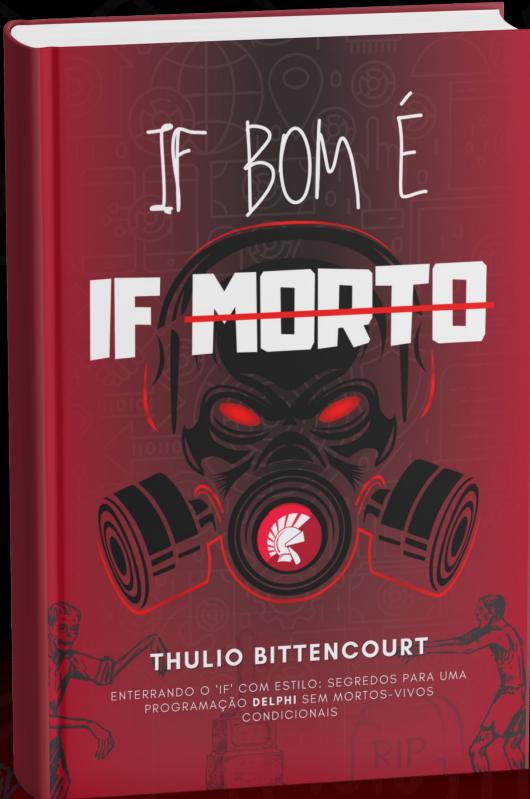


Francesco Cirillo



4 Livros sobre desenvolvimento de Software

Adquira os livros clicando aqui



Lançado em 2024, o livro "IF bom é IF morto" aborda a importância de minimizar o uso de condicionais IF na programação, mostrando como o excesso de IFs pode tornar o código complexo, difícil de manter e propenso a erros. Através de exemplos práticos e técnicas avançadas, o livro ensina como substituir IFs por alternativas mais eficientes, como polimorfismo, padrões de design e programação funcional. Com um enfoque em boas práticas e refatoração, o livro oferece um guia para desenvolver códigos mais limpos, claros e de fácil manutenção, que seguem os princípios da programação orientada a objetos e respeitam o desempenho do sistema.

Adquira em [@thuliobittencourt](#) (link na bio)

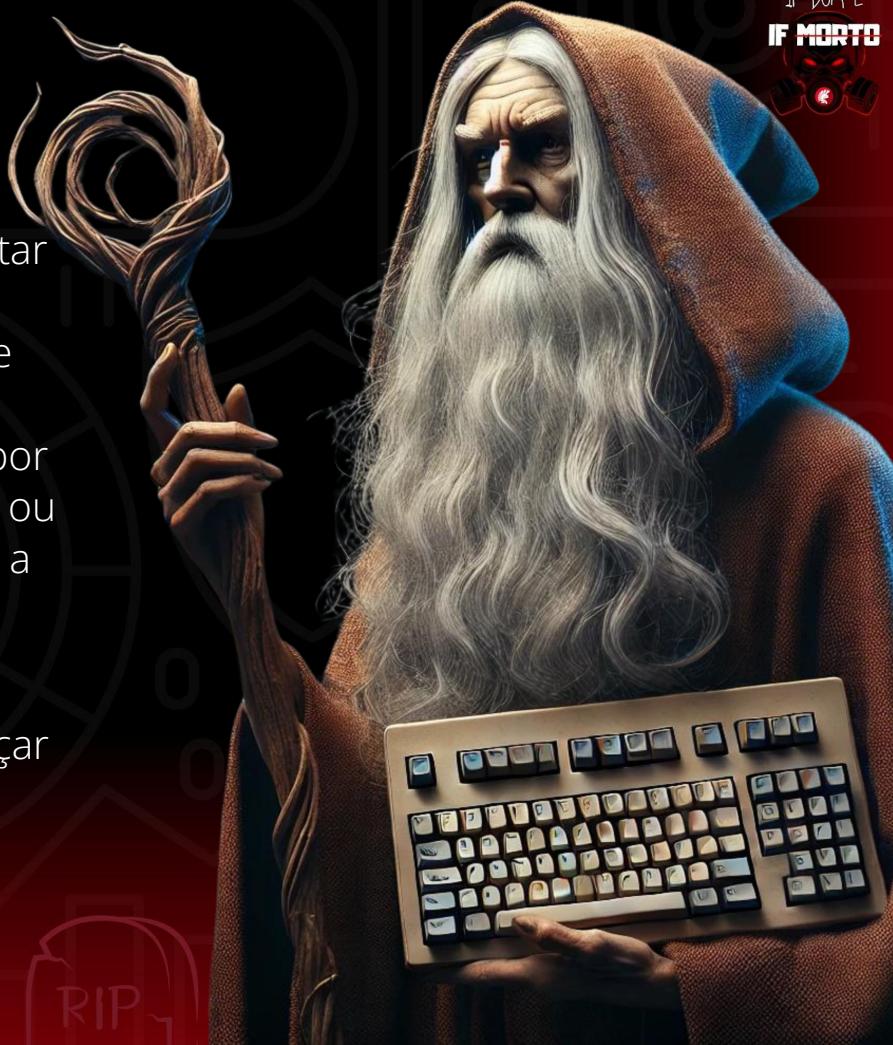
Cuidado com as entidades malignas do submundo da Programação Imperativa



Gandalf the Grumpy

Ele é um desenvolvedor antigo que se recusa a adotar novas tecnologias ou técnicas. Ele vive no passado, insistindo que tudo deve ser feito à moda antiga. Ele prefere códigos dentro do botão e procedures obscuras direto no banco de dados e é conhecido por sua aversão a qualquer coisa relacionada a DevOps ou automação. "Você não deve passar!" é sua resposta a qualquer tentativa de modernização.

Fraqueza: Frameworks modernos e qualquer coisa em nuvem. Apenas mencionar "Docker" o faz balançar seu bastão em fúria.





Agent Smith Legacy

Ele vê o código legado como uma matriz perfeita que deve ser preservada a todo custo. Ele considera qualquer tentativa de refatoração uma ameaça à "ordem natural" do código. Assim como Neo é caçado na Matrix, ele caça desenvolvedores que tentam modernizar seu código, sabotando suas tentativas de melhoria.

Fraqueza: Refatoração e Clean Code o fazem perder o controle, pois ele não consegue compreender um mundo sem suas "linhas perfeitas de código legado".

The Terminator of Spaghetti

Ele é um desenvolvedor que gera código espaguete com a precisão de uma máquina. Ele é implacável e não se importa com a manutenibilidade. Seu objetivo é apenas fazer o código funcionar, não importa o quanto confuso ou emaranhado ele fique. "I'll be back" é o que ele diz sempre que precisa corrigir algo em seu próprio labirinto de código.

Fraqueza: Ferramentas de análise estática e qualquer menção a padrões de design. Ele se torna desorientado quando confrontado com código limpo e bem estruturado.



Jabba the Hardcoder

Jabba the Hardcoder é preguiçoso e adora a facilidade de hardcoding. Ele engorda seu código com variáveis globais e com nomes que ninguém entende. Seu código é uma fortaleza de valores imutáveis, e ele exige que tudo seja feito à sua maneira. "**Você aprenderá a apreciar meus métodos codificados**" é sua filosofia.

Fraqueza: A simples ideia de renomear variáveis ou utilizar um ORM o deixa em pânico e desconfortável.



Daqui para
frente
somente com
proteção
divina



Oração a São Pascal

Ó Grande Pascal dos Céus,
Que nos guiou pelo caminho da
tipagem forte e das boas práticas.
Venho a Ti em busca de proteção,
Para que nenhum espírito maligno
Javiano se aproxime de mim. E tente
me seduzir com suas chaves {} e sua
verborragia sem fim.

Oração a São Pascal

Guarda-me, ô TTreeView e TStringList. Para que eu nunca caia na tentação de encapsular até minha xícara de café em uma classe. E que o poder do begin e end me mantenha longe das linhas intermináveis de código.

Ô. Sagrado TForm, guia meu cursor. Para que eu nunca precise de uma Factory para criar um simples botão. E afasta de mim a tentação de usar getters e setters para tudo. Deixando-me seguro no simples e glorioso :=

Oração a São Pascal

Que a luz do Delphi ilumine meu Ide. Para que eu nunca troque o conforto dos vcl pela prolixidade de outro paradigma.. E que meu código seja sempre limpo, eficiente e livre de exceções sem sentido.

Em nome de todos os Forms, Units, e Events,
Declaro: Ctrl+Shift+F9, e que meu código
compile sem erros! E, livrai-nos de todo
espírito javiano. Pois teu é o poder, a
performance, e a clareza. Hoje e para sempre.

Amém.

O
Problema
com IFs

Impactos do IF
na
Performance

Alternativas ao
IF

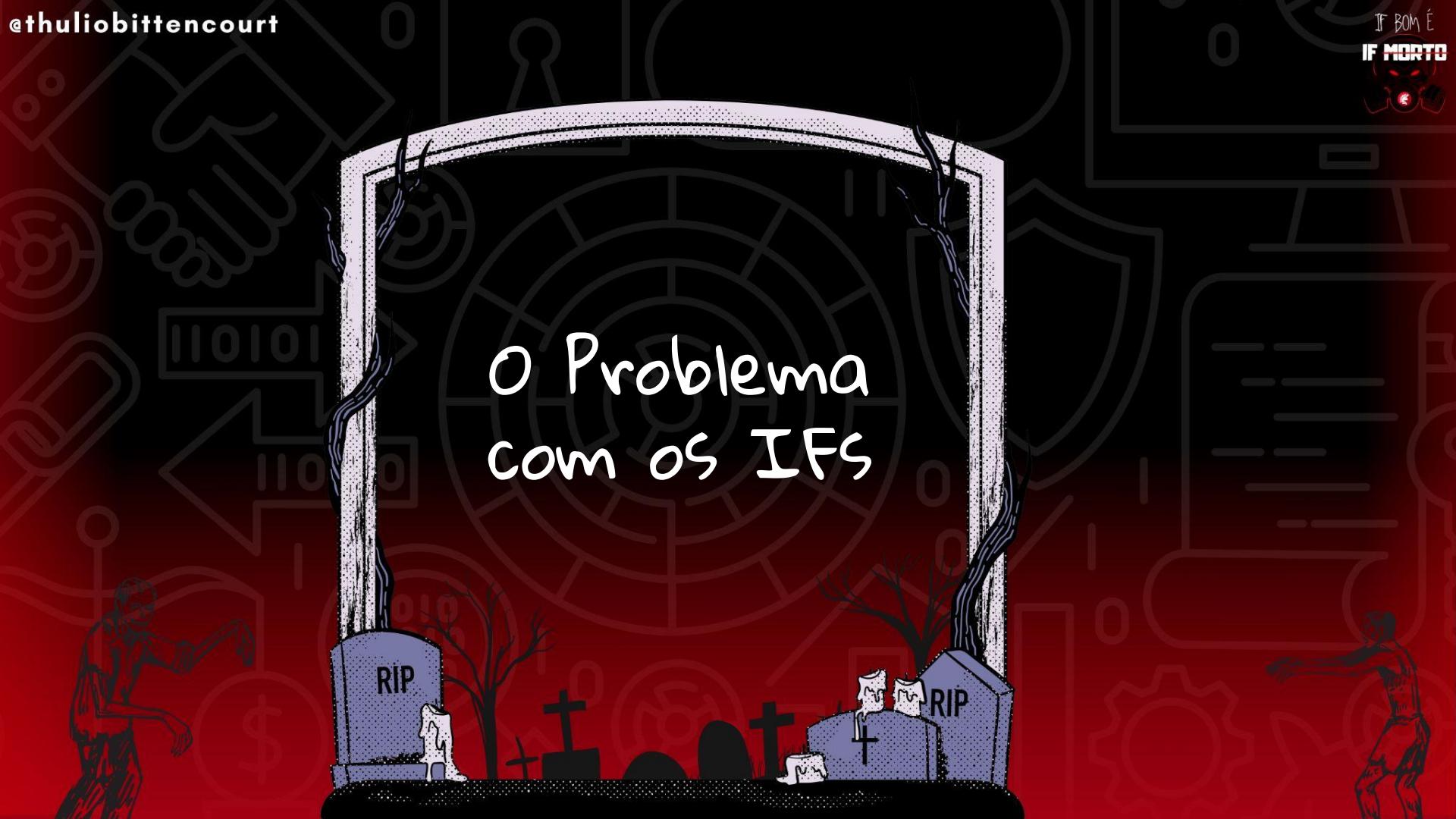
Técnicas de
Refatoração
para Reduzir
IFs

Programação
Funcional
como
Alternativa

Programação
Baseada em
Eventos

Refatorando
Sistemas
Legados

O Problema com os IFs



O IF SEMPRE VAI EXISTIR!

“Reducir o uso do IF não é sobre eliminar as condições, mas sim sobre escrever código mais inteligente, onde cada linha tem um propósito claro e cada decisão é feita com intenção.”

— Thulio Bittencourt

Por que minimizar o uso de IFs na programação

- **Complexidade e Manutenção:** O uso excessivo de IFs pode aumentar a complexidade do código, tornando-o difícil de entender, manter e refatorar.
- **Legibilidade:** Muitos IFs, especialmente quando aninhados, tornam o código confuso e difícil de seguir, aumentando a carga cognitiva para quem o lê.
- **Propensão a Erros:** IFs aumentam o número de caminhos lógicos, o que pode levar a mais bugs e dificuldades na garantia de cobertura de testes completa.
- **Impacto na Performance:** Em situações críticas, muitos IFs podem prejudicar a performance do código, especialmente em loops ou em operações sensíveis a tempo.

Paradigma Spaghetti Structured

Problemas com este código:

- Complexidade Aninhada:** O código tem muitos IFs aninhados, dificultando a leitura e o entendimento do fluxo.
- Manutenção Difícil:** Qualquer mudança em uma regra de negócio ou adição de um novo tipo de cliente, ou status exige modificações em vários pontos do código.
- Baixa Reutilização:** A lógica para processamento de pedidos está completamente acoplada, dificultando reutilizar partes do código sem copiar e colar.

```
procedure ProcessarPedido(statusPedido, tipoCliente: String;
valorPedido: Double);
begin
  if statusPedido = 'Pendente' then
    if tipoCliente = 'VIP' then
      if valorPedido > 1000 then
        AplicarDescontoEspecial(valorPedido)
      else
        ProcessarSemDesconto(valorPedido)
    else
      if tipoCliente = 'Comum' then
        if valorPedido > 500 then
          AplicarDescontoPadrao(valorPedido)
        else
          ProcessarSemDesconto(valorPedido)
      else
        ShowMessage('Tipo de cliente desconhecido!')
    else
      if statusPedido = 'Aprovado' then
        ShowMessage('Pedido já aprovado.')
      else
        if statusPedido = 'Cancelado' then
          ShowMessage('Pedido cancelado.')
        else
          ShowMessage('Status do pedido desconhecido!')
  end;
```

Como o uso excessivo afeta a clareza do código

- **Desvio Cognitivo:** Cada IF introduz uma bifurcação mental que o desenvolvedor precisa acompanhar, aumentando a carga cognitiva.
- **Perda de Intenção:** O excesso de IFs obscurece a intenção original do código, dificultando entender o fluxo lógico e a finalidade de cada trecho.
- **Aumento de Erros:** Com muitas condições a serem consideradas, é fácil perder de vista o que cada IF deveria controlar, aumentando a probabilidade de introduzir erros.

```
procedure ProcessarPagamento(tipoPagamento: String; valor: Double; clienteVIP: Boolean; possuiDesconto: Boolean);
begin
    if (tipoPagamento = 'cartao') or (tipoPagamento = 'boleto') then
        if (tipoPagamento = 'cartao') and (valor > 1000) or (tipoPagamento = 'boleto') and (valor > 500) then
            if clienteVIP then
                if possuiDesconto then
                    AplicarDesconto(valor)
                else
                    CobrarTaxa(valor)
            else
                if not possuiDesconto then
                    CobrarTaxa(valor)
                else
                    ShowMessage('Pagamento não aprovado.')
            else
                if clienteVIP and (valor <= 1000) then
                    CobrarTaxa(valor)
                else
                    ShowMessage('Valor abaixo do permitido ou pagamento não aprovado.')
            else
                ShowMessage('Tipo de pagamento desconhecido.');
        end;
    end;
```



Qual o antidoto universal anti-bugs?

Bugs se alimentam
de códigos, não
importa se tem
muitas ou poucas
linhas, dá para
fazer merda de
qualquer jeito



Manutenibilidade e dificuldade de teste

- **Código Rígido e Frágil:**

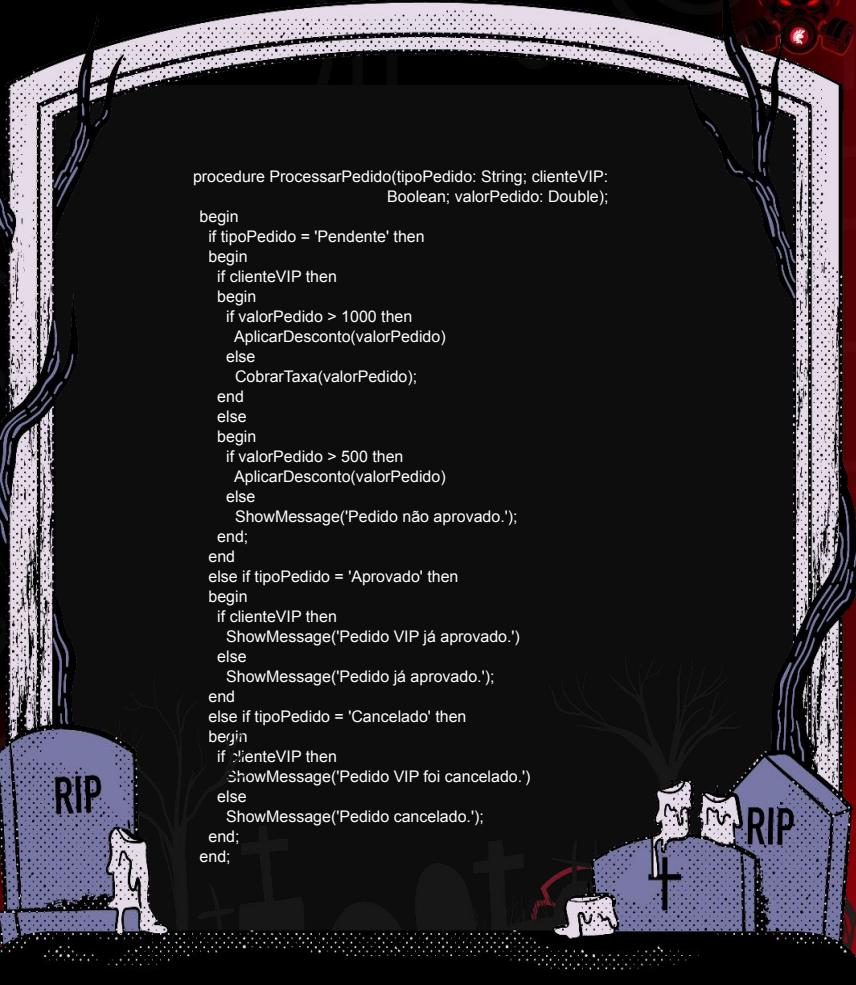
- O uso excessivo de IFs pode tornar o código rígido, dificultando a adaptação a novas funcionalidades ou mudanças de requisitos.
- Cada modificação no código pode desencadear uma cascata de mudanças necessárias, aumentando o risco de introduzir novos bugs.

- **Dificuldade na Testabilidade:**

- Muitos IFs criam múltiplos caminhos lógicos que precisam ser cobertos por testes, aumentando a complexidade e o esforço necessário para garantir que todos os cenários sejam verificados.
- A complexidade dos testes aumenta exponencialmente com a quantidade de IFs, dificultando a manutenção de uma suíte de testes robusta.

- **Impacto na Manutenibilidade:**

- Manter códigos com muitos IFs exige um esforço maior para entender e modificar o comportamento do software, especialmente em projetos grandes ou legados.
- A leitura e compreensão do código por novos desenvolvedores ou equipes diferentes se torna mais desafiadora, retardando o progresso e aumentando os custos de manutenção.



```
procedure ProcessarPedido(tipoPedido: String; clienteVIP: Boolean; valorPedido: Double);  
begin  
  if tipoPedido = 'Pendente' then  
    begin  
      if clienteVIP then  
        begin  
          if valorPedido > 1000 then  
            AplicarDesconto(valorPedido)  
          else  
            CobrarTaxa(valorPedido);  
        end  
      else  
        begin  
          if valorPedido > 500 then  
            AplicarDesconto(valorPedido)  
          else  
            ShowMessage('Pedido não aprovado.');//  
        end;  
    end;  
  else if tipoPedido = 'Aprovado' then  
    begin  
      if clienteVIP then  
        ShowMessage('Pedido VIP já aprovado.')  
      else  
        ShowMessage('Pedido já aprovado.');//  
    end;  
  else if tipoPedido = 'Cancelado' then  
    begin  
      if clienteVIP then  
        ShowMessage('Pedido VIP foi cancelado.')  
      else  
        ShowMessage('Pedido cancelado.');//  
    end;  
end;
```

TRADICIONAL

```
if (usuarioLogado) then
begin
  if (usuarioAdmin) then
  begin
    if (horaAtual > horaLimite) then
    begin
      // Faça algo
    end
    else
    begin
      // Faça outra coisa
    end;
  end
  else
  begin
    // Um caminho para usuários não-admin
  end;
end
else
begin
  // Usuário não logado
end;
```

REFATORADA

```
procedure ProcessarAcesso(usuario: TUsuario);
begin
  if not usuario.Logado then Exit;

  if usuario.Admin and (HoraAtual > HoraLimite) then
    RealizarOperacaoAdmin
  else
    RealizarOperacaoUsuario(usuario);
end;
```

APLICANDO O PRINCÍPIO DA RESPONSABILIDADE ÚNICA

```
type
  IOperacaoUsuario = interface
    ['{algum-GUID}']
    procedure Executar;
  end;

  TOperacaoAdmin = class(TInterfacedObject, IOperacaoUsuario)
    procedure Executar; override;
  end;

  TOperacaoComum = class(TInterfacedObject, IOperacaoUsuario)
    procedure Executar; override;
  end;

procedure TOperacaoAdmin.Executar;
begin
  // Lógica para admin
end;

procedure TOperacaoComum.Executar;
begin
  // Lógica comum
end;
```

```
procedure ProcessarAcesso(usuario: TUsuario);
var
  operacao: IOperacaoUsuario;
begin
  if usuario.Admin then
    operacao := TOperacaoAdmin.Create
  else
    operacao := TOperacaoComum.Create;

  operacao.Executar;
end;
```

REFATORANDO COM PADRÃO

TRADICIONAL

```
if (usuarioValido) then
begin
    if (senhaCorreta) then
        begin
            if (contaAtiva) then
                begin
                    // Permitir acesso
                end
            else
                begin
                    // Conta não está ativa
                end;
        end
    else
        begin
            // Senha incorreta
        end;
end
else
begin
    // Usuário inválido
end;
```

Definindo Estrutura Básica

```
type
  IAutenticacao = interface
    ['{algum-identificador-único}']
    procedure Executar; // Não retorna string
    procedure SetProximo(const Value: IAutenticacao);
    property Proximo: IAutenticacao write SetProximo;
  end;
```

Redefinindo Funções de Autenticação

```
procedure TUsuarioValido.Executar;
begin
  if not UsuarioEhValido then
    raise EUsuarioInvalido.Create('Usuário inválido');
end;

procedure TSenhaCorreta.Executar;
begin
  if not SenhaEhCorreta then
    raise ESenhaIncorreta.Create('Senha incorreta');
end;

procedure TContaAtiva.Executar;
begin
  if not ContaEhAtiva then
    raise EContaInativa.Create('Conta não está ativa');
end;
```

Tratamento de Exceções no Autenticador

```
class function TAutenticador.Autenticar(usuario: TUsuario): string;
var
  autenticacao, usuarioValido, senhaCorreta, contaAtiva: IAutenticacao;
begin
  try
    usuarioValido := TUsuarioValido.Create;
    senhaCorreta := TSenhaCorreta.Create;
    contaAtiva := TContaAtiva.Create;

    usuarioValido.Proximo := senhaCorreta;
    senhaCorreta.Proximo := contaAtiva;

    autenticacao := usuarioValido;
    autenticacao.Executar; // Lança exceção se falhar

    Result := 'Acesso Permitido'; // Sucesso se nenhuma exceção for lançada
  except
    on E: EUsuarioInvalido do Result := E.Message;
    on E: ESenhaIncorreta do Result := E.Message;
    on E: EContaInativa do Result := E.Message;
    // Outras exceções específicas podem ser adicionadas aqui
  end;
end;
```

Executando a autenticação

```
procedure TFormLogin.ButtonLoginClick(Sender: TObject);
var
    Usuario: TUsuario;
begin
    Usuario := TUsuario.Create;
    try
        Usuario.Nome := EditNomeUsuario.Text;
        Usuario.Senha := EditSenha.Text;

        try
            TAutenticador.Autenticar(Usuario);
            ShowMessage('Login bem-sucedido!');
        except
            on E: EUsuarioInvalido do ShowMessage(E.Message);
            on E: ESenhaIncorreta do ShowMessage(E.Message);
            on E: EContaInativa do ShowMessage(E.Message);
        end;

        finally
            Usuario.Free;
        end;
    end;
```

Impactos do IF na Performance



Quando o IF pode ser um problema de desempenho

- **Predição de Ramificação:**
 - Processadores modernos tentam prever o resultado dos IFs para otimizar a execução do código.
 - Previsões erradas causam atrasos, pois o processador precisa descartar operações e recalcular, prejudicando a performance.
- **Impacto em Loops:**
 - IFs dentro de loops críticos podem causar problemas significativos de desempenho, especialmente em operações de alta frequência ou em processamento de grandes volumes de dados.
- **Complexidade Condicional:**
 - IFs com múltiplas condições complexas aumentam o tempo de execução, pois o processador precisa avaliar cada condição antes de continuar.



Predição de ramificação

- **O Que é Predição de Ramificação:**

- Processadores modernos utilizam técnicas de predição de ramificação para "adivinar" o caminho que o código seguirá ao encontrar um IF.
- Isso permite que o processador continue executando instruções sem esperar pela resolução do IF, otimizando a performance.

- **Impacto de Previsões Erradas:**

- Quando a predição de ramificação falha, o processador precisa descartar as instruções executadas com base na predição incorreta e recalcular o caminho correto.
- Esse processo de correção (conhecido como "pipeline flushing") causa atrasos significativos, impactando negativamente a performance.



Predição de ramificação

- **Cenários Críticos:**

- Em loops ou códigos críticos onde muitas condições IF variáveis são encontradas, o risco de previsões erradas aumenta, causando degradação da performance.

- **Otimizando para Melhorar a Predição:**

- Simplificar ou reorganizar as condições IF para serem mais previsíveis pode ajudar a reduzir o impacto negativo.
- Alternativas como tabelas de pesquisa ou técnicas de programação funcional podem ser usadas para evitar IFs complexos.



Pipeline Flushing

- **O Que é Pipeline?**

- O pipeline é uma técnica usada por processadores modernos para executar várias instruções em paralelo, dividindo a execução em diferentes estágios (como busca, decodificação, execução, etc.).

- **Predição de Ramificação e Pipeline:**

- O processador tenta prever o caminho que o código seguirá (por exemplo, se uma condição IF será verdadeira ou falsa) para manter o pipeline cheio e funcionando eficientemente.

- **Pipeline Flushing:**

- Ocorre quando a predição de ramificação está errada, e o processador percebe que as instruções carregadas no pipeline são incorretas.

- Como resultado, o processador precisa "descarregar" (ou "flushing") todas as instruções erradas, invalidando-as e reiniciando o pipeline com as instruções corretas.



Writing a Simple Operating System —
from Scratch

by
Nick Blundell

School of Computer Science, University of Birmingham,
UK

Draft: December 2, 2010

Copyright © 2009-2010 Nick Blundell

Writing a Simple Operating System from Scratch - Nick Blundell

[Clique para baixar](#)

Casos de previsão correta e errada

CASO DE PREVISÃO CORRETA

```
procedure ProcessarDados(valores: array of Integer);
var
  i, total: Integer;
begin
  total := 0;
  for i := 0 to High(valores) do
  begin
    // Previsão correta: todos os valores são positivos
    if valores[i] > 0 then
      Inc(total, valores[i]);
    end;
    ShowMessage('Total: ' + IntToStr(total));
  end;
```

```
procedure TestePrevisaoCorreta;
var
  dados: array[0..4] of Integer = (10, 20, 30, 40, 50);
begin
  ProcessarDados(dados);
end;
```

CASO DE PREVISÃO ERRADA

```
procedure ProcessarDados(valores: array of Integer);
var
  i, total: Integer;
begin
  total := 0;
  for i := 0 to High(valores) do
  begin
    // Previsão errada: valores são positivos e negativos de
    // forma imprevisível
    if valores[i] > 0 then
      Inc(total, valores[i]);
    end;
    ShowMessage('Total: ' + IntToStr(total));
  end;
```

```
procedure TestePrevisaoErrada;
var
  dados: array[0..4] of Integer = (10, -20, 30, -40, 50);
begin
  ProcessarDados(dados);
end;
```

Alternativas ao IF



Uso de Polimorfismo

- **O Que é Polimorfismo:**

- Polimorfismo é um princípio da programação orientada a objetos que permite que diferentes classes respondam de maneira específica a uma mesma interface ou método base.
- Com polimorfismo, objetos de diferentes tipos podem ser tratados uniformemente, permitindo a substituição de comportamentos sem modificar o código que os utiliza.

- **Substituindo Condicionais Complexas:**

- Em vez de usar múltiplos IFs ou switch/case para lidar com diferentes tipos ou estados, o polimorfismo permite que cada classe implemente seu próprio comportamento específico.
- O código cliente invoca métodos na interface ou classe base sem precisar saber qual implementação específica está sendo usada.



@thuliobittencourt

IF BOM É
IF MORTO



Substituição por Polimorfismo

COM IF

```
procedure ProcessarPagamento(tipoPagamento: String; valor: Double);
begin
  if tipoPagamento = 'CartaoCredito' then
    begin
      // Processa pagamento com cartão de crédito
      ShowMessage('Processando pagamento de ' + FloatToStr(valor) + ' com Cartão de Crédito.');
    end
  else if tipoPagamento = 'Boleto' then
    begin
      // Processa pagamento com boleto
      ShowMessage('Processando pagamento de ' + FloatToStr(valor) + ' com Boleto.');
    end
  else if tipoPagamento = 'Pix' then
    begin
      // Processa pagamento com Pix
      ShowMessage('Processando pagamento de ' + FloatToStr(valor) + ' com Pix.');
    end
  else
    begin
      ShowMessage('Tipo de pagamento desconhecido.');
    end;
end;
```

SEM IF

```
procedure ProcessamentoComPolimorfismo;
var
  pagamento: IPagamento;
begin
  pagamento := TCartaoCredito.Create;
  ProcessarPagamento(pagamento, 100.0);
end;

procedure ProcessarPagamento(pagamento: IPagamento; valor: Double);
begin
  pagamento.Processar(valor);
end;
```

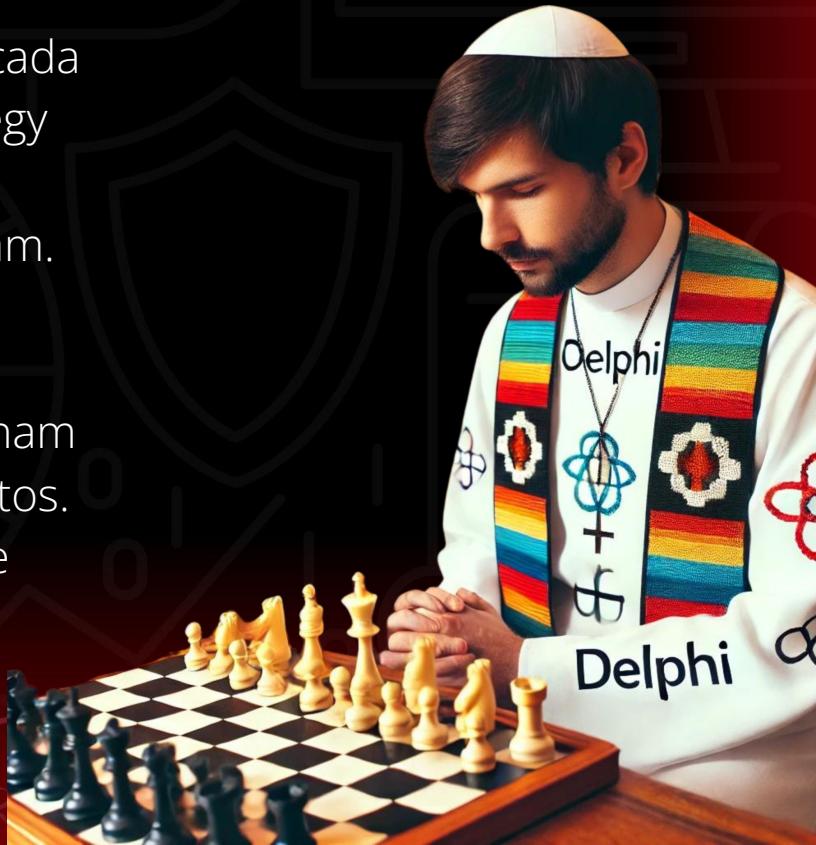
Padrões Strategy

- **O Que É:**

Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

- **Uso:**

Ideal para substituir condicionais que selecionam entre diferentes algoritmos ou comportamentos. Em vez de usar IFs, você troca dinamicamente entre diferentes estratégias.



COM IF

```
procedure CalcularDesconto(clienteTipo: String; valor: Double);
var
  desconto: Double;
begin
  if clienteTipo = 'VIP' then
    begin
      desconto := valor * 0.20; // 20% de desconto para clientes VIP
    end
  else if clienteTipo = 'Regular' then
    begin
      desconto := valor * 0.10; // 10% de desconto para clientes regulares
    end
  else if clienteTipo = 'Novo' then
    begin
      desconto := valor * 0.05; // 5% de desconto para novos clientes
    end
  else
    begin
      desconto := 0; // Sem desconto para outros tipos de clientes
    end;

  ShowMessage('O desconto é: ' + FloatToStr(desconto));
end;
```

SEM IF

```
ICalcudoDesconto = interface
  function Calcular(valor: Double): Double;
end;

procedure CalcularDescontoStrategy(cliente: ICalcudoDesconto; valor: Double);
var
  desconto: Double;
begin
  desconto := cliente.Cacular(valor);
  ShowMessage('O desconto é: ' + FloatToStr(desconto));
end;

procedure CalculoDescontoComStrategy;
var
  cliente: ICalcudoDesconto;
begin
  cliente := TDescontoVIP.Create;
  CalcularDescontoStrategy(cliente, 100.0);
end;
```



Padrão State

- **O Que É:**

Permite que um objeto altere seu comportamento quando o seu estado interno muda. O objeto parecerá mudar sua classe.

- **Uso:**

Substitui condicionais que verificam o estado de um objeto. Cada estado é encapsulado em uma classe separada, permitindo transições limpas entre estados sem IFs.

Substituição por State

COM IF

```
type
  TLampada = class
  private
    FEstado: String;
  public
    procedure Ligar;
    procedure Desligar;
    constructor Create;
  end;

procedure TLampada.Ligar;
begin
  if FEstado = 'Desligada' then
  begin
    ShowMessage('Lâmpada ligada.');
    FEstado := 'Ligada';
  end
  else
  begin
    ShowMessage('A lâmpada já está ligada.');
  end;
end;
```

SEM IF

```
TLampada = class;

IEstadoLampada = interface
  procedure Ligar;
  procedure Desligar;
end;

TLampada = class
  private
    FEstado: IEstadoLampada;
  public
    procedure SetEstado(Estado: IEstadoLampada);
    procedure Ligar;
    procedure Desligar;
    constructor Create;
  end;

procedure TLampada.Ligar;
begin
  FEstado.Ligar;
end;
```

Padrão Command

- **O Que É:**

Encapsula uma solicitação como um objeto, permitindo parametrizar métodos com diferentes solicitações, enfileirar ou registrar operações, e suportar operações reversíveis.

- **Uso:**

Elimina a necessidade de longas cadeias de condicionais para determinar qual ação executar. Com Command, cada ação é encapsulada em seu próprio objeto.



COM IF

```
type
TEditorTexto = class
public
procedure ExecutarComando(comando: String);
procedure Copiar;
procedure Colar;
procedure Desfazer;
end;

procedure TEditorTexto.ExecutarComando(comando:
String);
begin
if comando = 'Copiar' then
  Copiar
else if comando = 'Colar' then
  Colar
else if comando = 'Desfazer' then
  Desfazer
else
  ShowMessage('Comando desconhecido.');
end;
```

SEM IF

```
type
IComando = interface
  procedure Executar;
end;

procedure TestarEditorComCommand;
var
  Editor: TEditorTexto;
  Comando: IComando;
begin
  Editor := TEditorTexto.Create;
  Comando := TComandoCopiar.Create(Editor);
  Comando.Executar;
end;
```

Técnicas de Refatoração para Reduzir IFS





Simplificação de condições e Early Return

- **O Que É:**

- Early Return é uma prática onde você retorna de um método o mais cedo possível, ao invés de aninhar o código em blocos IF-ELSE.

- **Vantagens:**

- Reduz a complexidade do fluxo de controle, tornando o código mais linear e fácil de seguir. Também ajuda a evitar profundos níveis de aninhamento.

Substituição por Early Return

COM IF

```
procedure ProcessarPedido(pedidoValido: Boolean;  
estoqueDisponivel: Boolean);  
begin  
  if pedidoValido then  
    begin  
      if estoqueDisponivel then  
        begin  
          ShowMessage('Pedido processado com  
sucesso.');//  
        end  
      else  
        begin  
          ShowMessage('Estoque indisponível.');//  
        end;  
    end  
  else  
    begin  
      ShowMessage('Pedido inválido.');//  
    end;  
end;
```

SEM IF

```
procedure ProcessarPedido(pedidoValido:  
Boolean; estoqueDisponivel: Boolean);  
begin  
  if not pedidoValido then Exit;  
  if not estoqueDisponivel then Exit;  
  ShowMessage('Pedido processado com sucesso.');//  
end;
```

木 林 木 林

ÁRVORE BOSQUE FLORESTA

Tabelas de Pesquisa

- **O Que São Tabelas de Pesquisa:**

- Tabelas de pesquisa (lookup tables) são estruturas de dados que mapeiam entradas para saídas correspondentes diretamente, eliminando a necessidade de cálculos ou lógica condicional complexa durante a execução.

- **Quando Usar:**

- Substituição de Condicionais Complexas: Quando você tem várias condições ou IFs que mapeiam entradas específicas para saídas específicas, uma tabela de pesquisa pode simplificar o código.
 - Performance Crítica: Ideal em situações onde a performance é crítica, já que o acesso a uma tabela de pesquisa é geralmente constante, ao contrário de múltiplas verificações condicionais.



TRADICIONAL

```
procedure MostrarMensagemErro(codigoErro:  
Integer);  
begin  
if codigoErro = 404 then  
    ShowMessage('Erro: Página não encontrada.')  
else if codigoErro = 500 then  
    ShowMessage('Erro: Erro interno do servidor.')  
else if codigoErro = 403 then  
    ShowMessage('Erro: Acesso negado.')  
else if codigoErro = 200 then  
    ShowMessage('Sucesso: Operação concluída  
com sucesso.')  
else  
    ShowMessage('Erro desconhecido.');//  
end;
```

```
procedure TestarCodigoErroComIF;  
begin  
    MostrarMensagemErro(404);  
end;
```

REFATORADA

```
procedure MostrarMensagemErro(codigoErro:  
Integer);  
var  
    TabelaErros: TDictionary<Integer, String>;  
    mensagem: String;  
begin  
if TabelaErros.TryGetValue(codigoErro,  
mensagem) then  
    ShowMessage(mensagem)  
finally  
    TabelaErros.Free;  
end;  
end;
```

```
procedure TestarCodigoErroComTabela;  
begin  
    MostrarMensagemErro(404);  
end;
```

Programação Funcional como Alternativa



Programação Funcional como Alternativa

- **O Que é Programação Funcional:**
 - Um paradigma de programação que trata a computação como a avaliação de funções matemáticas, evitando mudanças de estado e dados mutáveis.
 - Enfatiza o uso de funções puras, imutabilidade e funções de ordem superior (funções que podem receber outras funções como argumentos ou retorná-las).
- **Principais Conceitos:**
 - **Funções Puras:** Funções que, para as mesmas entradas, sempre produzem as mesmas saídas, sem efeitos colaterais.
 - **Imutabilidade:** Dados não são modificados após a criação, reduzindo a complexidade e o risco de bugs relacionados a estados mutáveis.
 - **Funções de Alta Ordem:** Funções que podem aceitar outras funções como parâmetros, permitindo operações como map, filter, e reduce.
- **Vantagens em Relação ao Paradigma Imperativo:**
 - **Redução da Complexidade:** Menos dependência de estados mutáveis e IFs complexos.
 - **Legibilidade e Manutenibilidade:** Código mais claro e conciso, focado na transformação de dados em vez de em controlar o fluxo de execução.
 - **Facilidade de Teste:** Funções puras são mais fáceis de testar porque são determinísticas e não dependem de estados externos.



Mapeamento e filtragem em vez de condicionais

- **Mapeamento (map)**
 - É uma técnica onde você aplica uma função a cada elemento de uma coleção, transformando os elementos de acordo com uma lógica definida, sem a necessidade de IFs.
- **Filtragem (filter)**
 - É uma técnica onde você cria uma nova coleção contendo apenas os elementos que satisfazem uma condição específica, eliminando a necessidade de múltiplos IFs para selecionar itens.
- **Vantagens em Relação ao Uso de Condicionais:**
 - **Redução de Complexidade:** Mapeamento e filtragem reduzem a necessidade de múltiplos IFs aninhados, tornando o código mais limpo e fácil de entender.
 - **Legibilidade:** O código se torna mais declarativo, expressando diretamente a intenção ("transformar" ou "selecionar") sem depender de controle de fluxo manual.
 - **Reutilização:** As funções de mapeamento e filtragem podem ser reutilizadas em diferentes contextos, promovendo a modularidade do código.



TRADICIONAL

```
procedure ProcessarNumerosImperativo(valores: array of
Integer);
var
  i: Integer;
  resultados: array of Integer;
  count: Integer;
begin
  count := 0;
  SetLength(resultados, Length(valores));

  for i := 0 to High(valores) do
    begin
      if valores[i] mod 2 = 0 then
        begin
          resultados[count] := valores[i] * 2;
          Inc(count);
        end;
    end;

  // Exibir resultados
  for i := 0 to count - 1 do
    ShowMessage(IntToStr(resultados[i]));
end;
```

REFATORADA

```
procedure ProcessarNumerosFuncional(valores:
array of Integer);
var
  resultados: TArray<Integer>;
begin
  resultados := valores.Filter(EPar).Map(Dobrar);

  for var i in resultados do
    ShowMessage(IntToStr(i));
end;
```

Programação Baseada em Eventos



Programação Baseada em Eventos

- **O Que é Programação Baseada em Eventos:**
 - Um paradigma de programação onde o fluxo de execução é determinado por eventos, como cliques de mouse, teclas pressionadas, ou notificações de sistemas. Em vez de controle de fluxo tradicional (como IFs), o código responde dinamicamente a eventos conforme eles ocorrem.
- **Conceitos Básicos:**
 - **Eventos:** Ações ou ocorrências que o programa pode "ouvir" e reagir. Exemplo: o clique de um botão.
 - **Manipuladores de Eventos:** Funções ou métodos chamados automaticamente quando um evento específico ocorre. Exemplo: executar um procedimento ao clicar em um botão.
 - **Delegates (ou Ponteiros de Função):** Referências para métodos que podem ser passadas e chamadas dinamicamente, permitindo uma programação mais flexível e modular.
- **Vantagens em Relação ao Uso de Condicionais:**
 - **Desacoplamento:** A lógica do programa é separada do controle de fluxo, facilitando a modularidade e manutenção do código.
 - **Escalabilidade:** Adicionar novos comportamentos em resposta a eventos é simples e não exige alterações no código existente.
 - **Legibilidade e Manutenção:** O código se torna mais claro e fácil de seguir, pois se evita o uso de condicionais aninhadas para determinar qual ação tomar.



TRADICIONAL

```
procedure ProcessarMensagem(tipoMensagem: String; conteudo: String);
begin
  if tipoMensagem = 'Erro' then
    begin
      ShowMessage('Erro: ' + conteudo);
    end
  else if tipoMensagem = 'Aviso' then
    begin
      ShowMessage('Aviso: ' + conteudo);
    end
  else if tipoMensagem = 'Informacao' then
    begin
      ShowMessage('Informação: ' + conteudo);
    end
  else
    begin
      ShowMessage('Tipo de mensagem desconhecido.');
    end;
end;

procedure TestarProcessamentoComIFs;
begin
  ProcessarMensagem('Erro', 'Arquivo não encontrado');
end;
```

REFATORADA

```
type
  TMessageHandler = reference to procedure(conteudo: String);

procedure MostrarErro(conteudo: String);
begin
  ShowMessage('Erro: ' + conteudo);
end;

procedure ProcessarMensagemEventDriven(tipoMensagem: String;
                                         conteudo: String);
var
  Handlers: TDictionary<String, TMessageHandler>;
  Handler: TMessageHandler;
begin
  Handlers := TDictionary<String, TMessageHandler>.Create;
  try
    if Handlers.TryGetValue(tipoMensagem, Handler) then
      Handler(conteudo)
    else
      ShowMessage('Tipo de mensagem desconhecido.');
  finally
    Handlers.Free;
  end;
end;
```

Substituição por Eventos

TRADICIONAL

```
procedure ControlarTemperatura(temperatura: Integer);
begin
  if temperatura < 18 then
    begin
      ShowMessage('Ligando o aquecedor...');
    end
  else if temperatura > 26 then
    begin
      ShowMessage('Ligando o ventilador...');
    end
  else
    begin
      ShowMessage('Temperatura ideal, mantendo estado atual.');
    end;
end;

procedure TestarControlarTemperaturaComIFs;
begin
  ControlarTemperatura(16); // Ligando o aquecedor
  ControlarTemperatura(22); // Temperatura ideal
  ControlarTemperatura(28); // Ligando o ventilador
end;
```

REFATORADA

```
TacaoTemperatura = reference to procedure;
TControleTemperatura = class
  private
    FEventos: TDictionary<Integer, TAcaoTemperatura>;
  public
    procedure ControlarTemperatura(temperatura: Integer);
    end;

  procedure
    TControleTemperatura.ControlarTemperatura(temperatura:
    Integer);
    var
      acao: TAcaoTemperatura;
    begin
      for temperatura in FEventos.Keys do
        begin
          if FEventos.TryGetValue(temperatura, acao) then acao();
        end;
    end;

  Controle.ControlarTemperatura(16);
```

Refatorando Sistemas Legados



Refatorando Sistemas Legados

- Desafios de Sistemas Legados:
 - **Complexidade Acumulada:** Código antigo tende a ter muitas dependências, padrões obsoletos e complexidade desnecessária, dificultando a manutenção.
 - **Falta de Testes:** Sistemas legados muitas vezes carecem de uma suíte de testes adequada, aumentando o risco de introduzir bugs ao tentar refatorar.
 - **Dependências Cíclicas:** Código legado frequentemente tem módulos fortemente acoplados, dificultando isolar e modificar partes do sistema.



Refatorando Sistemas Legados

- **Estratégias de Refatoração:**
 - **Passos Pequenos e Incrementais:** Evite grandes reescritas. Refatore em pequenas etapas, testando rigorosamente cada alteração.
 - **Introdução de Testes:** Comece criando testes para o comportamento atual do sistema, garantindo que a refatoração não introduza regressões.
 - **Isolamento de Módulos:** Identifique e isole partes do código que podem ser refatoradas independentemente. Isso pode envolver a extração de métodos ou classes.
 - **Substituição Gradual de IFs:** Identifique onde os IFs podem ser substituídos por padrões de design (como Strategy, State, ou Command) para reduzir a complexidade.



Passo a passo para remover Complexidade em um projeto real

- **1. Identifique os IFs Problemáticos:**

- **Análise do Código:** Revise o código existente para identificar áreas onde os IFs são complexos, repetitivos ou dificultam a manutenção.
- **Critérios para Substituição:** Foco em IFs que lidam com múltiplas condições, estão espalhados por várias partes do código, ou estão associados a comportamentos repetitivos.

- **2. Introduza Testes Automatizados:**

- **Criação de Testes:** Antes de modificar o código, crie testes que garantam o comportamento atual. Isso ajuda a evitar regressões durante o processo de refatoração.
- **Cobertura de Testes:** Assegure-se de que todas as condições dos IFs sejam cobertas pelos testes, garantindo que o comportamento esperado seja preservado.

Passo a passo para remover Complexidade em um projeto real

- **3. Substitua Condicionais por Polimorfismo:**

- **Implementação de Interfaces:** Crie interfaces ou classes-base para encapsular os comportamentos que variam com os IFs.
- **Criação de Subclasses:** Implemente subclasses para cada condição específica que o IF manipulava. Essas subclasses devem implementar o comportamento correspondente.
- **Refatoração do Código:** Substitua os IFs pela lógica polimórfica, onde o código cliente chama métodos nas classes derivadas em vez de verificar as condições manualmente.

- **4. Refatore para Padrões de Projeto:**

- **Aplicação de Strategy, State, ou Command:** Identifique padrões de design que possam ser aplicados para substituir IFs, como o uso de Strategy para substituir lógica condicional que escolhe entre diferentes algoritmos.
- **Revisão e Refatoração:** Revise o código após a substituição, garantindo que ele siga os princípios SOLID e que as dependências sejam minimizadas.

Passo a passo para remover Complexidade em um projeto real

- **5. Teste e Valide as Alterações:**
 - **Execução de Testes Automatizados:** Após a refatoração, execute os testes para garantir que a funcionalidade original foi preservada.
 - **Testes Manuais:** Realize testes manuais em cenários críticos para garantir que as mudanças não introduziram problemas inesperados.
- **6. Documente as Mudanças:**
 - **Atualização da Documentação:** Registre as alterações feitas e as razões para a refatoração. Isso facilita a manutenção futura e o entendimento por outros desenvolvedores.
 - **Compartilhamento com a Equipe:** Comunique as mudanças para a equipe, destacando os benefícios e qualquer impacto potencial no desenvolvimento futuro.
- **7. Monitore e Otimize:**
 - **Monitoramento de Desempenho:** Após a implantação, monitore o sistema para garantir que a refatoração resultou em melhorias de desempenho ou manutenibilidade.
 - **Feedback Contínuo:** Colete feedback da equipe e dos usuários para identificar áreas adicionais que podem se beneficiar de refatoração.

Recapitulando a importância de minimizar IFs

- **Redução da Complexidade:**
 - **Menos Carga Cognitiva:** Minimizar IFs simplifica o fluxo do código, tornando-o mais fácil de entender e manter.
 - **Evita Aninhamento Excessivo:** Reduz a necessidade de condicionais aninhadas que tornam o código difícil de seguir e propenso a erros.
- **Melhoria na Manutenibilidade:**
 - **Código Modular:** Ao substituir IFs por técnicas como polimorfismo e padrões de design, o código se torna mais modular e fácil de modificar sem introduzir bugs.
 - **Facilidade de Atualização:** Novos comportamentos podem ser adicionados sem alterar o código existente, seguindo o princípio do aberto/fechado (Open/Closed Principle).
- **Aprimoramento da Testabilidade:**
 - **Testes Mais Simples:** Menos IFs resultam em menos caminhos lógicos a serem testados, facilitando a criação de uma suíte de testes robusta.
 - **Menos Riscos de Regressão:** Código mais limpo e organizado é menos suscetível a erros durante a manutenção e refatoração.
- **Melhoria de Performance:**
 - **Predição de Ramificação:** Minimizar IFs pode reduzir os problemas de predição de ramificação em processadores modernos, melhorando a performance do código em cenários críticos.
 - **Execução Mais Eficiente:** Evita verificações desnecessárias, tornando o código mais rápido e eficiente, especialmente em loops e operações frequentes.
- **Facilita a Evolução do Sistema:**
 - **Escalabilidade:** Sistemas com menos IFs são mais fáceis de escalar e adaptar a novos requisitos.
 - **Adaptação a Novas Tecnologias:** Um código bem estruturado, sem condicionais excessivas, facilita a integração com novas tecnologias e práticas modernas.

A professional portrait of a man with dark hair and a beard, wearing a dark suit jacket, resting his chin on his hand.

Obrigado

Me siga no Instagram



Avalie a Palestra



@thuliobittencourt