Código Limpo:

Habilidades Práticas do Agile Software

...

Capítulo 3: Funções

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Pequenas

- Funções devem ser pequenas, o máximo possível.
- Quanto menores, melhor.
- Em geral, não devem passar de 20 linhas.

MAS COMO FAZER ISSO?

Faça apenas uma coisa!

“As funções devem fazer uma coisa.
Devem fazê-la bem.
Devem fazer apenas ela.”

Faça apenas uma coisa!

Como é difícil definir o que é “uma coisa”, tente:

- Observar o nível da complexidade de indentação da função
- Ver tamanho de seus blocos
- Descrever o propósito em uma linha, que não deve ser muito grande

Se for possível dividir o objetivo dessa função, faça isso, o ideal é deixá-las únicas e indivisíveis como átomos que são parte de uma molécula.

Afinal, o objetivo das funções é decompor um conceito maior.

Faça apenas uma coisa!

Uma função deve ter apenas um objetivo a ser realizado:

```
public void mediaEaprovacao(double nota1, double nota2, double nota3, double nota4) {
    double media = (nota1 + nota2 + nota3 + nota4) / 4;

    if (media >= 7) {
        System.out.println("Aluno aprovado com média: " + media);
    } else {
        System.out.println("Aluno reprovado com média: " + media);
    }
}
```

A função acima tem dois objetivos: calcular a média e imprimir se o aluno foi reprovado ou aprovado.

Faça apenas uma coisa!

A função anterior pode ser dividida em duas:

```
public double calcularMedia(double nota1, double nota2, double nota3, double nota4) {
    return (nota1 + nota2 + nota3 + nota4) / 4;
}

public String verificarAprovacao(double media) {
    if (media >= 7) {
        return "Aluno aprovado com média: " + media;
    } else {
        return "Aluno reprovado com média: " + media;
    }
}
```

Dessa forma temos duas funções mais curtas e cada uma com seu objetivo único!

Otimizando IF

Os blocos if podem ser melhorados colocando uma função ao invés de escrever diretamente no código a sua condição de execução:

```
public static String verificarAprovacao(double media) {
   if (alunoAprovado(media)) {
       return mensagemAprovado(media);
   } else {
       return mensagemReprovado(media);
   }
}
```

Blocos e indentação: seções dentro de funções

A seguir é apresentado um exemplo claro a não ser seguido:

```
public String verificarAprovacao(String nome, double[] notas) {
   //calcula a média
   double media = 0.0;
   for (double nota : notas) {
       media += nota;
   }
   media /= notas.length;
   // imprime o resultado
   if (media >= 6.0) return nome + " está aprovado com média " + media;}
   else { return nome + " está reprovado com média " + media;}
}
```

Se uma função está organizada em seções por comentários, há um forte indício que ela está fazendo mais de uma coisa e deve ser dividida.

Blocos e indentação

Para aprimorar, basta dividir as seções da função anterior em outras funções!

```
public double calcularMedia(double[] notas) {
    double soma = 0.0;
    for (double nota : notas) {
        soma += nota;
    }
    return soma / notas.length;
}
public String imprimirResultado(double media) {
    return media >= 6.0 ? "aprovado" : "reprovado";
}
public String verificarSituacaoAluno(String nome, double[] notas) {
    double media = calcularMedia(notas);
    String situacao = imprimirResultado(media);
    return nome + " está " + situacao + " com média " + media;
}
```

Blocos e Indentação

- Evite estruturas aninhadas.
- Evite blocos grandes ou múltiplos blocos.
- Instruções (if, else, while...) devem ter apenas uma linha.
- Se essa linha for uma chamada de função, dê a ela um nome descritivo.
- A indentação não deve passar de dois níveis.

Essas dicas têm o intuito de facilitar a leitura e compreensão das funções.

Um nível de abstração por função

Para ajudar a confirmar se a função realmente só faz uma coisa, cheque se as instruções estão no mesmo nível de abstração.

Evite misturar níveis de abstração na mesma função, pois isso pode causar confusão:

- Nível Baixo: cálculos matemáticos simples ou manipulação de dados básicos.
- Nível Médio: formatação, transformação ou validação para manipular dados e retornar resultados mais complexos.
- Nível Alto: funções para avaliação ou tomada de decisão para determinar o resultado final, com base em condições ou regras específicas.

Regra descendente

O código deve ser lido de cima para baixo.

Funções devem ser organizadas hierarquicamente, de forma que funções de nível superior chamem as de nível inferior.

O nível superior é mais abstrato e delega tarefas às funções inferiores. O nível inferior é mais concreto e realiza as tarefas, retornando valores para as funções superiores.

Isso deixa o programa estruturado e fácil de entender.

Regra descendente

A seguir é apresentado um exemplo claro a não ser seguido:

```
public double calcularSalarioLiquido(double salario) {
   double imposto = salario * 0.20;
   double salarioLiquido = salario - imposto;
   return salarioLiquido;
}
```

Essa função que calcula o salário líquido e o imposto, deveria ser separada em duas funções.

Regra descendente

Essa função ficaria melhor assim, separada:

```
public double calcularSalarioLiquido(double salario) {
   return salario - calcularImposto(salario);
}

public double calcularImposto(double salario) {
   double imposto = salario * 0.20;
   return imposto;
}
```

A função de calcular salário tem nível superior e delega o cálculo do imposto para outra de nível inferior, que retorna o resultado a função chamadora.

Estrutura Switch

O uso excessivo de estruturas de switch deixa o código difícil de ler, entender e modificar.

O switch viola o princípio Open-Closed do S.O.L.I.D, que prega que o código deve estar aberto para extensão, mas fechado para modificação.

Com o switch, o código fica aberto para modificações sempre que opções são adicionadas, prejudicando a estabilidade e consistência do código ao longo do tempo.

Funções com switch são grandes, complexas, com muitos casos e muitas linhas dentro de cada caso, podendo levar a erros.

Estrutura Switch

Uma alternativa ao switch é o polimorfismo, no qual diferentes implementações de uma função podem ser usadas a partir de um tipo base. Isso traz algumas vantagens:

- Flexibilidade e extensibilidade do código
- Ausência de modificação de código já existente

Para alcançar essa solução, deve-se criar uma hierarquia de classes, cada uma com sua própria implementação, de forma a ter uma variação de comportamento.

Prefira polimorfismo aos comandos switch!

Nomes: use nomes descritivos!

"*Você sabe que está criando um código limpo quando cada rotina que você lê é como você esperava*" – Princípio de Ward

Dicas para escolher bons nomes de funções:

- Quanto menor e mais específica a função, mais fácil é criar seu nome
- Nome extenso é melhor que um pequeno e enigmático
- Revele a intenção do código
- Evite abreviações não convencionais
- Seja consistente nos nomes usados ao longo do código

Um nome claro pode ajudar a reestruturar o código de forma mais eficiente

Parâmetros

Muitos parâmetros podem deixar as funções difíceis de entender e de testar, pois haverá muitas combinações. Algumas dicas podem ajudar ao se projetar funções:

- Use o mínimo possível, tente não passar de quatro parâmetros.
- Se for necessário usar mais de quatro, é melhor agrupá-los usando tipos compostos.
- Dê aos tipos compostos nomes claros e descritivos.
- Tipos primitivos (int, float...) podem levar a erros de interpretação.
- Use apenas o necessário para sua função realizar seu objetivo.
- Mantenha um padrão de nomenclatura.

Parâmetros: Parâmetros Lógicos

Não passe booleanos como parâmetros!

- Péssima prática de programação.
- Mostra que a função faz mais de uma coisa (if parâmetro true... else).
- Opte por dividir a função em duas, uma para o caso true outra para o false.

Parâmetros

Funções Mônades (recebem um parâmetro):

- Pode ser uma função que pergunta sobre o parâmetro, como na função “isDigit()”
- Ou uma função que transforma e retorna o parâmetro, como “exemplo.lower()”

Dica do livro para esse tipo de função:

```
//Novo Builder com "transformação" no final
StringBuilder out = in.append("transformação");
```

Parâmetros

Funções Díades, são aquelas com dois parâmetros:

- Tente seguir a convenção esperado-obtido para testes de funções, no qual o primeiro parâmetro é o que se espera como resultado, e o segundo é o valor atual.

Funções Tríades, são aquelas com três parâmetros:

- Sua complexidade é bem maior, portanto exige muito mais atenção e cuidado para sua criação.

Parâmetros de Saída

Parâmetros de saída são variáveis passadas a uma função para armazenar valores de retorno:

- Evite usar parâmetros de saída, pois eles atrapalham a legibilidade.
- Para mudar o estado de um objeto, use uma função dele próprio, não uma função fora dele.
- Use exceções em vez de parâmetros de saída para lidar com erros e exceções.
- Prefira também retornar valores usando estruturas de dados.

Parâmetros de Saída

O uso parâmetro de saída, não deixa claro se a função anexa footer como rodapé ou anexa um rodapé ao footer, exemplo:

```
appendFooter(s);
```

É necessário consultar a declaração da função para entendê-la, e isso é ineficiente pois nos faz perder tempo.

Essa solução deveria estar implementada dessa forma:

```
report.appendFooter();
```

Efeitos Colaterais

Efeitos colaterais são mentiras, pois funções prometem fazer apenas uma tarefa, mas fazem outras sem que você perceba!

```
public boolean validaSenha(String usuario, String senha) {
   if (senhaIncorreta(senha)) {
       return false;
   } else {
       inicializaSessao();
       return true;
   }
}

private boolean senhaIncorreta(String senha) {//verificar se a senha está incorreta}

private void inicializaSessao() {//inicializar a sessão}
```

Efeitos Colaterais

No exemplo anterior, o nome da função implica na validação da senha e no retorno de um boolean, mas caso a senha esteja correta ele chama a função de inicializar sessão.

Isso é um efeito colateral! Quem acreditar no nome da função pode acabar perdendo todos os dados da sessão, dependendo da ordem que as funções forem chamadas.

É uma má prática de programação, pois modifica um valor ou estado que não está diretamente relacionado aos parâmetros da função ou variáveis locais.

Efeitos Colaterais

Qual seria a solução para esse problema de efeitos colaterais na função?

A FUNÇÃO DEVE FAZER APENAS UMA COISA!!!

Separação comando-consulta

Separar comando e consulta aumenta a coesão e a clareza do código, portanto a função deve realizar apenas uma das duas operações!

Comandos: ações e operações no sistema, como alterar o estado de uma entidade.

Consultas: recuperam informações e dados do sistema, sem modificar

```
public double obterEAtualizarSaldo(Conta conta, double valor) {
   double saldoAtual = conta.getSaldo();
   double novoSaldo = saldoAtual + valor;
   conta.setSaldo(novoSaldo);
   conta.save();
   return novoSaldo;
}
```

Confuso, não? Veja a seguir como melhorar esse código!

Separação comando-consulta

O exemplo de código a seguir separa comando e consulta:

```
//Comando - alterar o saldo de uma conta bancária
public void setSaldoConta(double novoSaldo) {
   this.saldo = novoSaldo;
}

//Consulta - obter o saldo de uma conta bancária
public double getSaldo(){
   return saldo;
}
```

Prefira Exceções

- Retornar códigos de erro viola a separação comando-consulta, pois os comandos são utilizados como expressões de comparação em estruturas condicionais.
- É ruim pois exige que o chamador da função lide imediatamente com o erro, e também o faz perder tempo com consulta de documentação.
- Usar exceções ao invés de códigos de erro é melhor, pois assim o código fica separado do tratamento de erro.

Prefira Exceções

Exceções são sinalizadoras de erros ou condições anormais em um programa, portanto em vez de um código de erro, uma exceção ajudará a tornar o código limpo.

Exemplos de casos para usar exceções:

- Divisão por zero
- Acesso a arquivo inexistente
- Erro em biblioteca de terceiros

Blocos Try-Catch

- São úteis para lidar com exceções em situações inesperadas.
- O uso excessivo pode tornar o código mais difícil de entender e manter.
- Devem ter escopo limitado para não aumentar desnecessariamente a complexidade.
- Devem conter código conciso para uma melhor visualização e compreensão.
- O ideal é que possuam suas próprias funções de tratamento de erro.

Tratamento de Erro

“As funções devem fazer uma coisa só.
Tratamento de erro é uma coisa só.”

Se a função já trata erros, então não deve fazer mais nada!

Evite Repetição

- Dificulta a manutenção, pois o código precisará ser alterado em vários lugares.
- Torna difícil de ler e entender.
- Funções devem encapsular funcionalidades que serão usadas muitas vezes.
- O uso de constantes ajuda a manter o código fácil de manter.

Programação Estruturada

É uma paradigma que preza pela clareza e organização do código e foca na criação de estruturas de controle simples e organizadas, como loops e condicionais.

Evita o uso de instruções de salto, como o *goto*, que podem tornar o código difícil de entender e manter.

Reforça a necessidade da modularização do código em funções e métodos, de modo que a funcionalidade possa ser reutilizada e mantida de forma mais fácil.

Processo de Construção: como escrever bem?

O processo é parecido com qualquer outro tipo de escrita:

1 - Coloque os pensamentos num papel: ideias, o que você precisa, o que quer fazer.

2 - Organize-os de forma a ficarem fáceis de ler.

3 - Faça um primeiro rascunho.

4 - Refine, aperfeiçoe, organize o código, teste.

5 - Se houver funções grandes, divida-as em menores.

Faça isso quantas vezes for necessário até que a função esteja do jeito que você queira

Conclusão

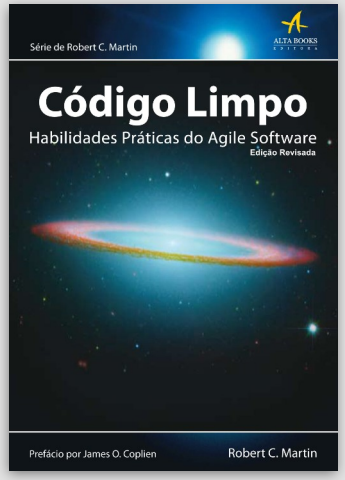
- O código limpo torna o software fácil de entender, manter e evoluir.
- Fazer código limpo **exige disciplina e prática,** alcançadas por meio do uso de princípios e práticas de codificação comuns.
- Veja códigos como histórias a serem contadas ao invés de programas a serem escritos.
- Código limpo aumenta a produtividade, reduz o tempo gasto e melhora a qualidade do software entregue.
- Código limpo é essencial para os DEVS que desejam criar software de alta qualidade, de fácil manutenção e que evolua ao longo do tempo.

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:** Habilidades Práticas do Agile Software. 1. ed. Rio de Janeiro: Alta Books, 2009.



A equipe

Fábio Seyiji, Autor

Aluno e bolsista do PET/ADS
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn