Clean Code

– CÓDIGO LIMPO

```
function formatName(name){
  if (name) {
    return name.trim();
  }
 }
}
```

JEAN BIERRENBACH

Instituto Fourier Digital

Código Limpo

Mini Curso - Boas Práticas de Programação

[width=4cm]logo_fourier.png

Jean Bierrenbach
Instituto Fourier Digital

Inspirado na obra de Robert C. Martin

28 de junho de 2025

Código Limpo

Boas Práticas para Desenvolvedores Profissionais

Jean Bierrenbach
Instituto Fourier Digital

1ª Edição

Inspirado na obra "Clean Code"de Robert C. Martin

Sumário

Prólogo

"Código limpo é música que ressoa em cada linha."

Bem-vindo ao **Clean Code Mini Curso**! Neste material expandido, exploramos **10 capítulos** de boas práticas de engenharia de software, cada um com explicações aprofundadas, exemplos práticos em Java e dicas de mestre para elevar seu código ao nível profissional.

Este trabalho foi inspirado na obra seminal de Robert C. Martin e adaptado à realidade dos desenvolvedores brasileiros pelo Instituto Fourier Digital.

1. Comentários Bem-Utilizados

Neste capítulo, você aprenderá quando **valem** comentários no código e quando **inferem** má qualidade. Comentários devem:

- Explicar algoritmos complexos (ex: TOTP, criptografia)
- Documentar decisões arquiteturais críticas
- Alertar sobre armadilhas ou efeitos colaterais inesperados

Listing 1.1: Comentário explicativo útil

```
public final class TotpAuthenticator {
    private static final int TIME_STEP_SECONDS = 30;
    //        Alterar invalida tokens existentes

    /**
    * Gera um TOTP de 6 d gitos (RFC 6238).
    * Passos:
    * 1. T = (tempo atual - T0) / TIME_STEP_SECONDS
    * 2. HMAC-SHA1(secret, T)
    * 3. Dynamic Truncation para obter d gitos
    */
    public String generate(String base32Secret) {
        long counter = (System.currentTimeMillis()/1000L)/
           TIME_STEP_SECONDS;
        int hash = (base32Secret + counter).hashCode();
        int otp = Math.abs(hash % (int)Math.pow(10, DIGITS));
        return String.format("%0" + DIGITS + "d", otp);
    }
}
```

2. Formatação

A **formatação** do código gera clareza instantânea. Você verá:

- Organização vertical com espaços em branco entre conceitos
- Quebras horizontais para evitar scrolling lateral
- Indentação consistente e nomes de variáveis legíveis

```
1  public class OrderService {
2
3      public void process(Order order) {
4          // Valida  o
5          if (order == null) {
6              throw new IllegalArgumentException("Order␣required"
                   );
7          }
8
9          // C lculo do total
10         for (Item item : order.getItems()) {
11             if (item.getQuantity() > 0) {
12                 BigDecimal lineTotal = item.getPrice()
13                     .multiply(BigDecimal.valueOf(item.
                           getQuantity()));
14                 order.addTotal(lineTotal);
15             }
16         }
17
18         // Notifica  o
19         notifyCustomer(order);
20     }
21
22     private void notifyCustomer(Order order) {
23         // TODO: enviar e-mail
24     }
25 }
```

3. Objetos e Estruturas de Dados

Diferencie **objetos** (comportamento + encapsulamento) de **estruturas de dados** (DTOs, sem lógica). Isso impacta na coesão e na manutenção do sistema.

```
// Objeto com comportamento
public class BankAccount {
    private BigDecimal balance;

    public void deposit(BigDecimal amount) { ... }
    public void withdraw(BigDecimal amount) { ... }
}

// Estrutura de dados (DTO)
public class BankAccountDTO {
    public BigDecimal balance;
    public String owner;
    public String accountNumber;
}
```

4. Tratamento de Erros

Erros devem ser tratados de forma **clara** e **previsível**:

- Fail Fast: detectar entrada inválida imediatamente
- Exceções Checked vs Unchecked
- Mensagens descritivas e limitadas a blocos mínimos de try-catch

```
public void transfer(Account from, Account to, BigDecimal
   amount) {
    if (from == null || to == null) {
        throw new IllegalArgumentException("Contas␣
           obrigat rias");
    }
    if (amount.compareTo(BigDecimal.ZERO) <= 0) {
        throw new IllegalArgumentException("Valor␣deve␣ser␣>␣0"
           );
    }
    // l gica de d bito/cr dito...
}
```

5. Limites e Adapter

Use o padrão **Adapter** para criar uma fronteira (boundary) entre seu domínio e APIs externas. Isso mantém seu código desacoplado e facilmente testável.

```
public interface SecurePaymentGateway {
    PaymentResponse processPayment(PaymentRequest request);
}

public class ThirdPartyAdapter implements SecurePaymentGateway
   {
    private final ExternalService svc;

    public PaymentResponse processPayment(PaymentRequest req) {
        // Adapta  o entre nossa interface e API externa
        ExternalPaymentRequest extReq = convertRequest(req);
        ExternalPaymentResponse extResp = svc.process(extReq);
        return convertResponse(extResp);
    }
}
```

6. Injeção de Dependência Avançada

A **Inversão de Controle (IoC)** e a injeção de dependência permitem:

- Configuração centralizada (@Configuration do Spring)
- Scopes e testes com beans falsos (mocks, fakes)

```
@Configuration
public class AppConfig {
    @Bean
    public AccountRepository repo(DataSource ds) { ... }

    @Bean
    public FraudDetector detector() { ... }

    @Bean
    public TransferService service(
        AccountRepository r,
        FraudDetector d
    ) {
        return new TransferService(r, d);
    }
}
```

7. Design Emergente

Quatro regras para design que **emerge** naturalmente:

1. Todos os testes devem passar
2. Sem duplicação (DRY)
3. Nomes expressam a intenção
4. Minimizar número de classes e métodos

```
public class ShoppingCart {
    public BigDecimal calculateTotal() {
        // Implementa  o coesa e sem duplica  o
    }

    public void applyDiscount(BigDecimal rate) {
        // Comportamento claramente nomeado
    }
}
```

8. Refinamento Sucessivo

Refatore código legado em **pequenos passos**, sempre protegidos por testes de caracterização:

- Extrair parser, validador e transformador
- Manter comportamento até o fim

```
public class DataProcessor {
    private final FileParser parser;
    private final DataValidator validator;
    private final DataTransformer transformer;

    public void process(DataFile f) {
        // Processamento em etapas claras
        RawData raw = parser.parse(f);
        ValidData valid = validator.validate(raw);
        Result result = transformer.transform(valid);
        return result;
    }
}
```

9. Refatoração com Segurança

Use **testes de caracterização** e substitua implementações antigas por APIs robustas:

- Exemplo clássico: addDays com Calendar e LocalDate
- Técnica de Strangler Fig para substituição progressiva

```
// M todo legado (deprecated)
public static Date addDays(Date date, int days) { ... }

// Nova implementa  o com java.time
public static LocalDate addDays(LocalDate date, int days) {
    return date.plusDays(days);
}
```

10. Heurísticas Avançadas

Aplicação das heurísticas G16, G28, G31, G34 para:

- Melhorar clareza de condicionais
- Evitar acoplamento temporal
- Manter um nível de abstração por método

```
// Condicional clara com m todo expressivo
if (isEligible(txn)) {
    processTransaction(txn);
}

// M todo com n vel  nico  de abstra  o
public boolean isEligible(Transaction t) {
    return hasSufficientFunds(t)
        && withinDailyLimit(t)
        && isValidCurrency(t);
}
```

Índice Remissivo