02 - Meaningful Names

- Nomear é parte intrínseca do processo
- Nomes devem mostrar a intenção.
    - E alterá-los se houver um nome melhor
    - Deve dizer o quê, o como e porquê daquela variável.
    - Idealmente um nome deve dispensar comentários
- Código deve ser explícito.
    - Nomes não devem suscitar mais perguntas
- Mesmo código simples se torna difícil de entender se houver muitas informações implícitas
  NÃO DESINFORMATIVO
- Se atentar para nomes que podem ter vários significados
    - e.g. uma variável "lista" é realmente um data type lista ou só uma listagem metafórica?
        - De qualquer modo é melhor evitar data type no nome.
    - Evitar nomes longos difícil de ler à primeira vista.
    - Evitar l e O, confunde com 1 e 0.
      NÃO NÃO-INFORMATIVO
- Não basta nomes de variáveis serem diferentes, eles têm que ter significados diferentes.
    - Não é bom criar uma variável quase igual a outra só diferente por um número ou uma grafia.
    - e.g. a1 e a2. *ProductInfo* & *ProductData*
- As diferenças devem ter significado

PRONUNCIÁVEL

- É bom usar nomes pronunciáveis.
    - e.g. não usar cstmr para customer.
      BUSCÁVEL
- Importante usar nomes buscáveis
    - e.g. evitar a, i, j, k.
- Escrever nomes mais longos se necessário.
    - Nomes de uma letra só podem ser usados no máximo apenas em funções curtas em escopo local.

NÃO-CODIFICADO

- Ser explícito no que a variável faz. Evitar notação húngara.
- Evitar member prefix ("m_") para atributos de classe.
- Evitar I de interface.

- Ser claro, não usar abreviações ou inferências mentais para saber o que uma variável faz.
- Devem ser um substantivo.
    - Evitar palavras como Manager, Processor ou Info no nome.
- Métodos devem ter um verbo.

NOMES DE VARIÁVEIS

CLASSES

- Não ser fofo ou engraçadinho. Ser claro e direto.
- Usar sempre o mesmo verbo para um dado significado.
    - e.g. não misturar get com fetch e retrieve.
    - Controller & manager & driver - Tudo bem ter vários métodos em classes diferentes com o mesmo nome se conceitualmente eles fizerem a mesma coisa.
    - e.g. método *add*
    - Mas só os métodos add forem semanticamente iguais. e.g. Somar elementos e inserir um elemento na lista não podem os dois serem chamados de *add* - Tudo bem usar termos técnicos da programação - Domínio das Soluções
    - e.g. job, queue
- Quando não der, usar os termos do problema - Domínio dos Problemas
    - e.g. Customer.
- Adicionar contexto pode aumentar claridade, quando já não for claro.
    - e.g. Usar addrFirstName em vez de FirstName, quando se tratar de um endereço
    - Uso de classes pode dispensar isso se o contexto já for evidente
- Nomeação precisa ser descritiva
- Não devemos ter medo de renomear variáveis por nomes melhores.

01 - Clean Code

INTRODUÇÃO

- Código sempre será necessário - detalhes das especificações está nele
- Código bom não é supérfluo, é essencial.
    - Necessário para conserto e expansão
- Lei de LeBlanc: "Depois é igual a nunca"
- Código ruim é rápido no começo mas se torna devagar com o tempo.
    - Impossível de manter, baixa produtividade.
- É importante defender na organização a importância do código bom.
    - Desenvolvedor é quem detém o conhecimento da importância do código
    - Não deve se submeter à pressão do cronograma.
    - Código ruim é mais lento no longo prazo.
- Aprender a escrever código limpo requer prática e experiência.
    - Senso intuitivo de o que é um código limpo ou não.

O QUE É CÓDIGO LIMPO?

- Há muitas definições de código limpo.
    - Elegante
    - Eficiente
    - Claro
    - Focado
    - Direto
    - Simples
    - Legível
    - Testado
    - Mínimo
- Há várias escolas de pensamento sobre código. Esta é uma.
    - Há discordâncias.
- Lemos mais do que escrevemos
    - Ler rápido e fácil faz parte do bom código
- Regra do escoteiro: Deixar o código mais limpo do que quando encontrou
    - Pequenas mudanças incrementais

03 - Functions

Como fazer uma função ser fácil de entender?

- Funções devem ser **bem pequenas.**
- 20 linha já é bem alto.
    - Ideal é até umas 4 linhas
- If statemente devem ser simples
    - Sem ifs aninhados.
    - No máximo duas indentações no código
- Funções devem fazer **uma coisa apenas**
    - Funções servem pra quebrar operações maiores, afinal
- Um nível de abstração por função
    - e.g. não juntar algo abstrato como "getHtml" com funções básicas como ".append('\n')"
- Leitura de funções deve ser fluida e um passo levar ao outro intuitivamente.
  SWITCH
- Switch statements requerem por definição várias linhas e fazem várias coisas ao mesmo tempo.
- Ideal é que fique escondido dentro de uma abstract factory.
    - Deve aparecer só uma vez
    - Deve criar objetos polimórficos
    - Escondidos atrás de inheritance
    - Exemplo:

Employee and Factory

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

NOMES

- Usar nomes bem descritivos para as funções.
  - Mesmo que o nome seja longo.
  - Escolher bons nomes pode requerer tempo, esforço e retentativas.
  - Ser consistente com nomes, não misturar termos equivalentes.

ARGUMENTOS

- Quantos menos argumentos, melhor.
  - Zero é o ideal (Niládico).
    - Seguido de Monádico (1) , Diádico (2)
  - Três é o limite, não é ideal. (Triádico)
  - Evitar funções poliádicas.
- Mais argumentos também dificulta testar as funções.
  - Mais combinações de input possíveis.
- É melhor evitar modificação do input in place (output argument).
  - A ação da função deve ser apenas pelo return value, não pela modificação do input.

MONÁDICOS

- Formas monádicas em relação ao argumento podem ser:
  - Extração
  - Transformação
  - Geração de eventos -- esse aqui deixar bem claro quando é o caso.
  - Melhor evitar outras formas.
- Argumentos flag (boolean) são uma péssima ideia.
  - Indica que uma função faz mais de uma coisa.
  - Melhor dividir em duas funções, uma para cada caso.

DIÁDICOS

- Dois argumentos são mais difíceis de entender do que um.
- Mas em alguns casos são naturais.
  - e.g. definir um ponto num plano cartesiano, x e y.

        - São duas informações de um mesmo dado.
    - Os dois argumentos devem ser coerentes entre si.

TRIÁDICO

- Bem mais difíceis de entender.
- Só devem ser criados quando são realmente necessários.
- Pode ser útil agrupar argumentos em um único objeto específico e usar esse objeto como argumento.
- Funções que recebem número variável de argumentos (e.g. print) são equivalentes a monádico, diádico ou triádico, é como se fosse uma lista só. Não são um problema em si.

- Funções monádicas costumam ter nome da forma <verbo>_<substantivo>.

SIDE EFFECTS

- Função não deve ter side-effects
    - i.e. ação fora do return value.
    - Gera ofuscação
- Também gera acoplamento temporal entre duas ações.
    - Se houver acoplamento temporal, o nome deve descrever isso
- Output arguments são desnecessários hoje.
    - Se uma função for mudar o estado de algo, que seja o do objeto que detém o método (self).

COMMAND QUERY SEPARATION

- Função deve fazer algo ou responder algo, nunca ambos.
- Por exemplo, função que faz algo e retorna um boolean de sucesso é ruim:
- Melhor usar a alterantiva mais clara e separada:
- É melhor usar exceptions do que códigos de erro por causa disso.
    - Error codes geram problema de dependências.
    - Corpo do Try / except devem ser funções mínimas por si só, para separar ações de lidar com erro e ação de fazer a ação esperada.
        - Para não violar o princípio de uma coisa por função.
        - Função não deve ter nada além do try /except / finally

|
|

```
public boolean set(String attribute, String value)
```

```
<br>if (attributeExists("username")) { <br>setAttribute("username",
"unclebob"); <br>... <br>}<br>
```

- Evitar duplicação
- Diminui lugares que requerem reescrita em caso de alteração.
- Análogo a normalização de banco de dados
- Muito importante!
- Toda função deve ter apenas uma entrada e uma saída.

    - Apenas um único RETURN
    - Sem BREAK, CONTINUE em loops
    - Sem GOTO

- Mas como as funções já são pequenas, tudo bem quebrar um pouco essas regras de vez em quando. (Exceto o GOTO)

- Não é trivial escrever código assim e requer treino.

    - Melhora deve ser um processo iterativo de refino.
    - Dificilmente alguém consegue escrever com essas regras desde o início.

DON'T REPEAT YOURSELF

ESTRUTURAÇÃO

04 - Comments

- Comentários são um mal necessário
    - Compensam o que o código não foi capaz de dizer sozinho
    - São sempre um resultado de falha na comunicação
- Podem ficar defasados em relação ao código de verdade.
- Comentários ruins são pior do que nenhum comentário
- Comentários não consertam código ruim
- Exemplo:

```
// Ruim:
   // Check to see if the employee is eligible for full benefits
   if ((employee.flags & HOURLY_FLAG) &&  (employee.age > 65))

// Bom:
        if (employee.isEligibleForFullBenefits())
```

COMENTÁRIOS BONS

- **Comentários legais:**
    - Copyright ou autoria.
- **Comentários informativos:**
    - Descreve o que está sendo feito, embora seja melhor o código ser autoexplicativo
    - Explicar Regex
- **Explicar intenção:**
    - Justificar decisões.
- **Esclarecimento:**
    - Explicar o significado de um valor do código que não podemos alterar nós mesmos
    - Pode ser um risco de estar errado.
- **Alerta:**
    - Avisar que uma função está desligada porque demora muito ou pode apagar algo.
- **TODO:**
    - Descrever tarefas futuras.
    - Não justifica deixar código ruim.
- **Ênfase:**
    - Destacar importância de alguma operação.

COMENTÁRIOS RUINS

- **Murmúrio:**
    - Comentário mal explicado sem sentido.
- **Redudante:**
    - Só repete o que o código já diz por si só sem acréscimo.
- **Enganoso:**
    - Descreve errado o que o código faz
- **Obrigatório:**
    - Exigir um docstring sempre é uma péssima ideia que só polui o código.
- **Registro:**
    - Registro das alterações ao longo do tempo, como um diário.
    - Já foram úteis mas hoje com Git não são mais.
- **Ruído:**
    - Não acrescenta nada. Completamente ignoráveis.

- **Substituição de um bom nome de funções e variáveis:**
  - O código deve falar por si só quando possível
- **Marcadores visuais / Banner:**
  - Desnecessários e poluição visual.
- **Explicação de colchete:**
  - Indicam função muito longa
  - Ifs e whiles e trys devem ser autoexplicativos e fáceis de seguir.
- **Autoria de modificação:**
  - Git já diz quem fez qual alteração, não precisa deixar escrito.
- **Código comentado:**
  - Outros vão ficar com medo de deletar, ficam se acumulando.
  - Não deve existir. Git armazena código antigo para nós.
- **Comentários HTML:**
  - Às vezes usado para renderizar bonito em outra ferramenta.
  - Não deve existir. A outra ferramenta que se encarregue de renderizar certo.
- **Informação não-local:**
  - Comentário deve sempre explicar o código adjacente.
  - Nunca deve explicar coisas do sistema inteiro.
- **Informação demais:**
  - Contextualização histórica.
  - Muitos detalhes, como uma página de um manual.
- **Conexão não-óbvia:**
  - Comentário não pode ser difícil de entender.
  - Comentário não deve precisar de um comentário para explicar a si próprio.
- **Título de função:**
  - Nome e ação da função deve ser autoexplicativo.
- **Docstring em códigos não-públicos:**
  - Desnecessário para código de consumo interno, formalidade a mais só atrapalha.

05 - Formatting

2025-10-03 22:22 pm

- Formatação demonstração cuidado com o código. - Regras devem ser consistentes. - Objetivo é comunicação, não rigidez. **FORMATAÇÃO VERTICAL**

- Qual o tamanho ideal de um arquivo?
    - De 200 a 500 linhas parece bom, mas depende.
- Arquivos menores são mais fáceis de entender.
- Pensar código como um jornal
    - Não ter nenhum artigo muito longo cheio de detalhes.
    - Separar em seções (arquivos)
- Linhas em branco ajudam formatação e separação entre conceitos.
- Blocos visuais de código devem indicar unidades lógicas no código.
- Afinidade conceitual:
    - Conceitos próximos entre si devem estar verticalmente próximos entre si.
    - Não devem existir em outros arquivos se são próximos.
    - Evitar que leitor salte muito entre pedaços de código.
- Variáveis:
    - Variável deve ser declarada o mais perto de seu uso possível.
        - No topo de uma função (que deve ser curta por sua vez)
        - Variável de loop deve ser declarada adjacente ao loop
    - Atributos de instâncias devem existir no topo da classe.
        - Algumas convenções colocam na parte mais embaixo.
        - Importante é ser padronizado.
- Funções dependentes:
    - Funções que chamam uma à outra devem existir próximas.
    - Em sequência, se possível. Do alto nível ao baixo nível.

FORMATAÇÃO HORIZONTAL

- Linhas curtas são melhores.
- Há vários limites sugeridos: 80, 100, 120.
    - Mais de 120 é excessivo.
- Densidade:
    - Espaço para dissociar componentes e não-espaço para indicar coesão
    - Atribuição de vaiável deve ter espaço em branco
    - Funções não devem ter espaço entre primeiro parêntese e argumento.
    - Operações matemáticas também deve ter espaço para clareza.
- Alinhamento:
    - Alguns padrões usam alinhamento em colunas (com vários espaços em branco para isso) de termos coesos:

```
public FitNesseExpediter(Socket          s,
                         FitNesseContext context) throws Exception
{
  this.context =            context;
  socket =                  s;
  input =                   s.getInputStream();
  output =                  s.getOutputStream();
  requestParsingTimeLimit = 10000;
}
```

    - Mas é melhor não fazer isso.
        - Melhor deixar desalinhado mesmo (i.e. um só espaço em branco separando os elementos na linha).
    - Se for uma lista longa de elementos que peçam alinhamento, o problema é o tamanho da lista, não o desalinhamento.
- Indentação:
    - Serve para explicitar hierarquia de scopes.
        - e.g. o que está fora e o que está dentro de um if/else.
    - É tentador violar a indentação para funções/loops curtos e deixar numa linha só.
        - Isso é uma má ideia.
        - É melhor ter indentação mesmo assim.

REGRAS DO TIME

- Formatação envolve preferências
- Estilo deve ser acordado e ser único para o time.

06 - Objects and Data Structure

Importante distinguir o que deve ser alterado e o que não, dentro de uma classe

ABSTRAÇÃO DE DADOS

- Distinguir o que e quando deve ser atribuido.
- Quem tem acesso a qual atributo.
- Podemos forçar que atributos sejam definidos simultaneamente com um setter.
- Concreto X Abstrato
    - Ideal é esconder implementação.
    - Requer cuidado e atenção para fazer isso. Não é só ter getters e setters.

ESTRUTURA DE DADOS X OBJETOS

- Estrutura de Dados e Objeto são opostos:
- Objeto esconde dados atrás de abstrações para exprimir uma funcionalidade
- Estrutura de Dados expõe os dados sem ter nenhuma funcionalidade explícita.
- Dicotomia:
    - Código procedural (código que usa estrutura de dados):
        - Facilita adicionar novas funções sem alterar estrutura de dados
        - Dificulta adicionar novas estrutura de dados pois requer alteração nas funções
    - Orientação a objeto:
        - Facilita adicionar novas classes/estrutura de dados sem alterar funções.
        - Dificulta adicionar novas funções porque as classes devem ser alteradas.
    - Escolha entre os dois depende da situação. Trade-off.

LEI DE DEMETER

- Módulo deve ser ignorante sobre os detalhes do objeto que ele manipula.
- "Não conversar com estranhos".
- Um método de uma classe C só deve chamar os métodos de:
    - A classe C em si
    - Objeto criado pelo método
    - Objeto argumento do método
    - Objeto em um atributo da classe C
- Exemplo de violação da regra:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

//Esse encadeamento de funções é ruim
//Melhor separar em várias linhas, como em:

Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

- Mesmo assim pode ser uma violação, dependendo da estrutura.
    - Ctxt, Options e ScratchDir são objetos ou estruturas de dados?
    - Se forem Estruturas de dados tudo bem expor sua estrutura interna. Se objeto, não
    - Dot notation seria melhor do que accessors aqui para sinalizar que é esturutra de dado.

HÍBRIDOS

- Objetos que são tanto objeto quanto estrutura de dados.
- Têm funcionalidade significativa e expõe os dados
- Pior dos dois mundos, devem ser evitados.

ESCONDER ESTRUTURA

- Pode requerer novos métodos para esconder dados internos, ser uma função.

DATA TRANSFER OBJECT (DTO)

- Estrutura de dados mais pura
- Apenas variáveis públicas, zero funcionalidade.
- Bem útil e frequente.

ACTIVE RECORD

- Tipo de DTO
- Tem dados públicos mas alguns métodos de navegação como save e find.
- Não deve conter regras de negócio.
    - Estas devem existir em outro objeto.

07 - Error Handling

Error Handling não deve confundir ou obscurecer a lógica do código.

Recomendações

- Usar Exceptions em vez de Return Codes:
  - Separa melhor funcionalidade do código.
  - Mais fácil do que lidar com o erro dentro do próprio corpo da função.
- Escrever o try-catch-finally antes de tudo:
  - Mais controle de o que pode acontecer com o código.
  - Facilita o desenvolvimento e a compreensão do fluxo do código.
  - Iterativamente: Forçar Exceções --> Lidar com elas.
- Usar Unchecked Exceptions:
  - Característica do Java (ausente no Python):
    - Checked excePtions: exceções checadas na compilação. Devem ser tratadas.
    - Unchecked exceptions: exceções checadas na execução. Podem ou não ser tratadas.
  - Checked exception violam princípio open/close.
    - Requerem declaração na assinatura -> ruim.
    - Pode gerar necessidade de criar uma cascata imensa de alterações.
- Dar contexto com as exceções.
  - Exceção deve prover informação sobre origem do erro.
  - Mensagens de erro devem ser informativas.
- Definir Classes de exceção a partir das necessidades do caller
  - Definição da exceção deve depender de como ela é capturada.
  - Não da fonte ou um tipo arbitrário.
  - Evitar vários catch em sequência para erros parecidos com o mesmo tratamento.
    - Melhor envelopar tudo numa única classe de exceção
    - É comum uma única classe ser o suficiente para uma região do código.
- Definir o fluxo normal
  - Evitar colocar parte da lógica dentro do tratamento de erro
  - Tratamento de casos especiais pode ser gerido por uma classe (Special Case Object)
  - Assim o código original não precisa lidar com o caso especial dentro da exceção.
- Não retornar nulo
  - None é uma fonte comum de erros
  - É ruim ter vários checks para None e é fácil esquecer um.
  - Em vez de retornar nulo é melhor virar uma exception ou retornar um Special Case Object.
- Não passar nulo
  - Passar None como argumento é pior ainda
  - Existem várias formas de lidar com argumentos nulos e nenhuma é boa.
  - Melhor é prevenir para nunca ter uma situação em que None é passado.

08 - Boundaries

2025-09-25 7:30 am

- Como lidar com código de terceiros que pode ter funcionalidades diferente do que queremos?
- É bom encapsular objetos comuns em objetos específicos com as especificações para o seu caso.
    - Detalhes de implementação ficam ocultos.
    - Interação se dá apenas pela classe específica.
- Entender e integrar uma biblioteca de terceiros pode ser difícil:
    - Usar testes - "Testes de aprendizado"
    - Criar testes unitários com casos de uso simples e partir daí para o uso de fato.
    - Ajuda no aprendizado e torna código mais robusto de qualquer forma.
    - Se protege de novos updates que quebram algo. - Escrever código para interagir com outro código que ainda será desenvolvido é também desafiador.
    - Mesmo sem saber a interface:
        - Escrever o código como seria a interface ideal imaginada.
    - No fim, basta escrever um adaptador para ligar a interface ideal criada à real. - Essas fronteiras entre código devem acomodar **mudanças** de forma fácil.
    - Requerem testes e separação clara de atribuições.
    - Assim pelo menos temos previsibilidade da parte que nós controlamos.
    - Menos pontos de falha e manutenção.

09 - Unit Tests

2025-09-25 8:13 am

- Testes devem ser permanentes, completos e facilmente reprodutíveis.

TRÊS LEIS DO TDD

1. Você não deve escrever código em produção até ter escrito um teste correspondente em falha
2. Você não deve escrever em um teste além do que o suficiente para falhar - inclusive falha por compilação
3. Você não deve escrever mais código em produção além do que é o suficiente para passar o teste em falha. 5. Teste e código de produção são escritos juntos.
4. Assim a cobertura será 100%.
    - Mas geri-los pode se tornar um problema por si só

MANTER TESTES LIMPOS

- Testes ruins são igual ou pior do que não ter testes.
    - Passam a falhar não se sabe por quê.
    - Vira um fardo.
- Testes devem seguir padrão e serem organizados.
- Código de teste é tão importante quando código de produção.
    - Requer tanto cuidado e atenção quanto.
- Testes dão flexibilidade ao desenvolvedor
    - Permite criar mudanças sem receios.

TESTES LIMPOS

- O que são?
- Mais importante**: LEGIBILIDADE**
    - Depende das mesmas coisas que tornam código de produção legível.
        - Clareza, simplicidade, etc.
    - Não deve ter detalhes desnecessários
- Padrão BUILD-OPERATE-CHECK de teste.
    - Três etapas simples.
- Uma alternativa é criar API específica para testes
- Podemos também criar uma linguagem para teste:
    - e.g. maiúsculo = ON, minúsculo = OFF

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
  }
```

DUPLO PADRÃO

- Código de teste não precisa ser eficiente.
    - Ambiente de teste não é o mesmo de produção
    - O que é ruim em um pode ser aceitável no outro
    - Mas ambos devem ser limpos

UM ASSERT POR TESTE

- Ideal é um assert por teste, mais fácil de ler
    - Se tiver vários, podemos dividir em vários testes
    - Mas às vezes isso gera código duplicado
- "given-when-then" convention:
    - Análogo ao build-operate-check
- Template Method:
    - Tenta evitar esta duplicação
- Podemos colocar o "given-when" em uma classe e criar os testes individuais apenas para o "then"
- Mas se for necessário, tudo bem colocar vários asserts.
    - O importante é que seja o mínimo necessário.
- O mais importante é ter UM CONCEITO POR TESTE

F.I.R.S.T.

- **Fast**:
    - Teste deve ser rápido, para rodar com frequência
- **Independent**:
    - Resultado de um teste não deve depender de outro, apenas dele em si. Sem efeito cascata.
- **Repeatable**:
    - Deve ser repetível em qualquer ambiente - seja de produção, de QA, no laptop, na rede principal, etc. Não deve depender do ambiente de forma alguma.
- **Self-Validating**:
    - Output deve ser Passar ou Falhar. Sem detalhes e sem requerer análise ou pensamento subjetivo.
- **Timely**:
    - Teste unitário deve ser escrito logo antes do código de produção que o faz passar. Código de produção portanto deve ser pensado para ser testável.

10 - Classes

2025-10-03 17:13 pm

- Por convenção classes devem começar com uma lista de variáveis
    - Nesta ordem:
        - Constantes públicas
        - Constantes privadas
        - Variável de instância privada.
    - Seguido de funções públicas
        - Funções privadas ficam listadas conforme são usadas
- Expor variáveis para testes é uma má ideia
    - Devem ser testados, mas por meio de encapsulação.
- Classes devem ser pequenas!
    - Medido em "responsabilidades"
    - Nome da classe deve descrever responsabilidades
        - Deve ser conciso
        - "Processor", "Manager" ou "Super" são maus sinais
    - Descrição da classe deve ser breve e sem "e", "se", "mas" ou "ou"
    - **Single Responsibility Principle (SRP)**
        - Uma classe ou um módulo deve ter uma e apenas uma razão para mudar.
        - Uma única responsabilidade (e.g. não juntar "gerir versão" com "mostrar dashboard")
        - É melhor ter muitas classes pequenas que poucas classes imensas
    - **Coesão**
        - Classe deve ter poucas variáveis
        - Quanto mais variáveis um método manipula, maior a coesão do método à classe
            - Coesão máxima: todas as variáveis são usadas em todas os métodos
        - Queremos coesão alta
            - Significa que fazem parte da mesma lógica
        - Se tem algum conjunto de variáveis que só é usado por alguns métodos, costuma significar que aquilo podia se tornar outra classe separada
        - Quando classes têm baixa coesão, convém dividi-las em classes menores.
        - - Organizar para mudanças
    - Mudança é contínua
    - Escrever código que requeira poucas alterações e tenha poucos pontos de falha
    - **Open-Closed principle (OCP)**
        - Classes devem ser tais que permitem extensão mas não modificação.
        - Novos features não devem requerer mudar nada, só adicionar.
    - **Dependency Inversion Principle (DIP)**
        - Classes devem depender de abstrações, não de detalhes concretos.

12 - Emergence

2025-10-03 17:08 pm

Queremos regras que naturalmente gerem boas diretrizes de design, como **SRP** e **DIP**

Regras de um bom design simples:

1. **Executa todos os teste**
2. **Não tem duplicações**
3. **Expressa a intenção do programador**
4. **Minimiza o número de classes e métodos**

1 - Executa todos os testes

- Design deve permitir sistema completamente testável
- Incentiva classes pequenas e de responsabilidade única (SRP) - facilita teste.
- Acoplamento excessivo dificulta testes - incentiva DIP e outros.
- Com os testes podemos pensar em refatoração e melhorias, como mudar nomes, diminuir acomplamento, etc. Regra 1 permite as regras 2 a 4.

2 - Não tem duplicações

- Duplicação é trabalho e risco a mais desnecessários

```
#Métodos diferentes similares
int size() {}
boolean isEmpty() {}

# Eliminamos uma duplicação juntando uma à outra
boolean isEmpty() {
return 0 == size()
}
```

- Template Method pattern é uma alternativa para evitar duplicações

3 - Expressivo

- Maior custo de software é manutenção
- Quanto mais claro, menos tempo para entender.
- Nomes claros, funções pequenas, usar padrões de nomenclatura e de design (e.g. Visitor).
- Bons testes também são expressivos
- Não basta ter o código só funcionando, tem que estar claro - inclusive para nós mesmos no futuro

4 - Classes e Métodos Mínimos

- SRP pode induzir excesso de pequenas classes - isso é ruim
- Importante evitar dogmas
- Esta é a regra menos importante das quatro.

11 - Systems

2025-10-03 16:09 pm

- Como manter padrão clean a um nível mais alto, mais abstrato. - Separar construção do uso
    - Separação do startup e da operação de um software.
        - Princípio da separação de responsabilidades
    - Startup não deve depender da implementação
        - Uso de "main" para essa separação
        - Uso de Abstract Factory

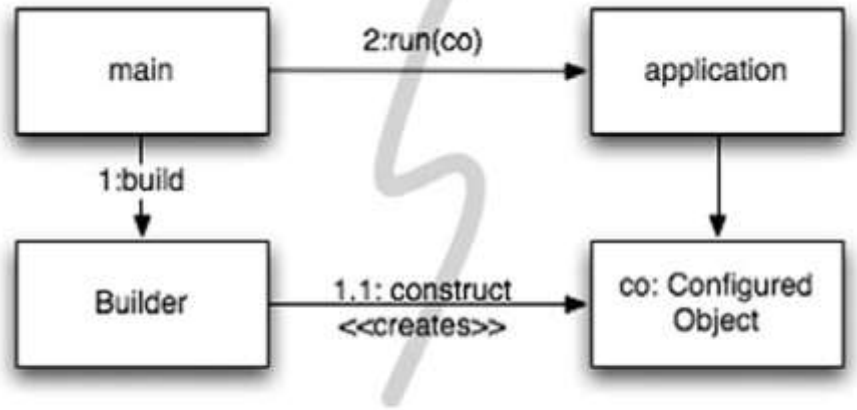


Figure 11-1

Separating construction in `main()`

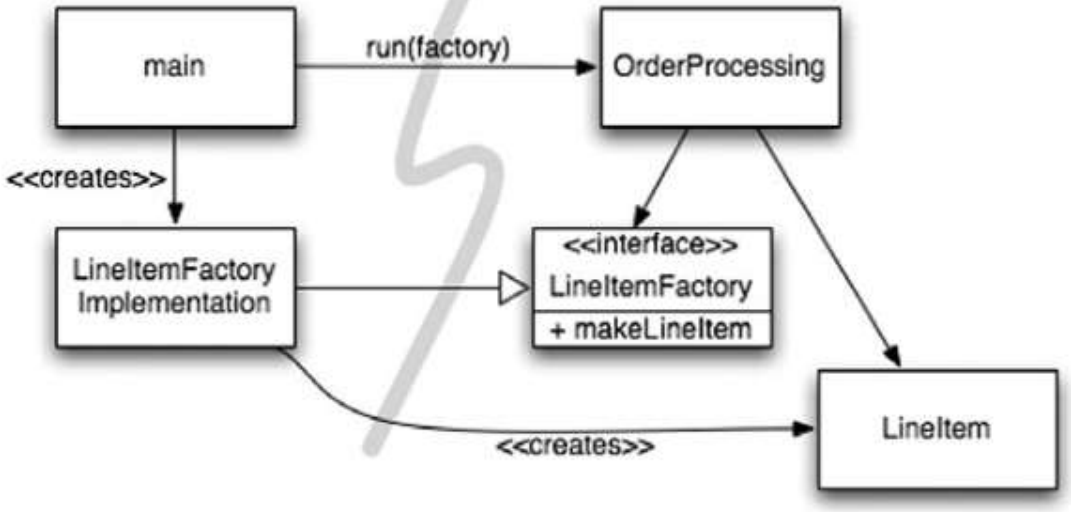


Figure 11-2

Separation construction with factory

    - Dependency Injection (DI) separa construção do uso.
        - Inversion of Control (IOC) aplicado à gestão de dependências
        - Classe recebe na instanciação os objetos de que precisa pra funcionar.
            - De preferência recebe os objetos conforme a necessidade.

Escalabilidade

- Não tem como fazer sistema grande e ótimo de código desde o começo

- Bom evitar acoplamento a objetos grandes.
- Cross-cutting Concern
    - Alguns aspectos são gerais em um software, como a forma de persistência
    - Deve ser característica adicionada ao código de forma não-invasiva.
    - *Aspect Oriented programming (AOP)*:
        - Algumas formas de fazer isso em Java:
            1. Java Proxies
            2. Frameworks AOP puros
            3. AspectJ Aspects
        - Em python dá pra fazer isso com decorators - separar cross-cutting concern da lógica de negócio
        - Esta forma de organizar o código é altamente recomendada
            - Evita **BDUF** (*Big Design Up Front*)
            - Facilita alterações e crescimento.
            - Facilita teste
- "*Uma boa arquitetura tem domínios modularizados de responsabilidades*"

```
# `The aspect (logging) is separate from the business logic`
def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}...")
        result = func(*args, **kwargs)
        print(f"{func.__name__} finished.")
        return result
    return wrapper

# Business logic is clean
@log_execution
def process_data(data):
    print(f"Processing: {data}")

process_data("Sample Data")
```

Otimização de Tomada de Decisões

- Modularização deste forma permite tomada de decisões decentralizadas
- É sempre bom adiar tomada de decisões até o último momento possível
    - Para ter a melhor informação
- Padrões só fazem sentido se eles adicionarem valor
- **DSL** *(Domain Specific Language)*
    - pequenas linguagem de scripting para facilitar comunicação
    - Um nível de abstração acima que pode tornar sistema mais explícito

"*Use the simplest thing that can possibly work*"

13 - Concurrency

2025-10-03 17:45 pm

- Escrever código com *Concurrency* limpo é muito difícil
- Concurrency desacopla **o quê do quando
    - Em single-thread, estão acoplados e é fácil de saber o estado do sistema
    - Desacoplamento melhora estrutura e velocidade do código.
- **Mitos:**
    - ❌ Concurrency sempre melhora o desempenho
        - Pode ou não melhorar
    - ❌ Design não muda quando escrevemos código com concurrency
        - Muda muito
    - ❌ Entender concurrency não é importante quando se lida com um container como Web Frameworks
        - É bom entender como funcionam para evitar problemas
- **Questões:**
    - Concurrency gera overhead de tempo de código
    - Concurrency correta é complexo
    - Bugs de Concurrency nem sempre são reproduzíveis
    - Concurrency requer mudanças fundamentais no design

Desafios:

- Há mais de um caminho possível para o mesmo código
    - Resultados diferentes possíveis

Princípios

- Há alguns princípios para proteger código desses problemas
- **SRP**:
    - Código de concorrência pode se misturar mas requer tratamento específico
    - Recomendação é separar concurrency do resto do código
    - **Limitar escopo dos dados:**
        - *synchronized* keyword permite um único thread por vez para proteger alguma seção crítica de dado compartilhado por threads (em Java)
            - Equivalente em Python seria usando um *Lock* com um *with*
    - **Usar cópias dos dados**:
        - Evitar compartilhamento de dados entre threads
        - Tratar variáveis como read-only e gerar cópias
    - **Threads devem ser o mais independentes possível:**
        - Thread deve existir por si só sem depender de nada externo a ela
    - *Ideal é tentar dividir os dados em subconjuntos independentes*

Recursos

- Recomendações:

    1. **Use thread-safe collections**

        - e.g. em Java usar `ConcurrentHashMap` em vez de `Hashmap`
    2. **Use o framework executor para realizar tarefas não relacionadas**
    3. **Use soluções não-bloqueantes quando possível**
    4. **Várias classes não são thread-safe**
    - Modelos de execução:
        - **Producer-Consumer**:
            - Thread producer insere tarefas numa queue e um consumer thread adquire aquela tarefas da fila e a executa.
            - Queue é um recurso limitado, tem um tamanho máximo.
            - Requer coordenação entre consumer e producer para agir no tempo certo.
        - **Reader-Writer:**
            - Reader não pode ler algo que o writer está escrevendo e vice-versa.
            - Writer bloqueia os readers.
            - Precisa equilibrar os dois para evitar starvation de um ou do outro.
        - **Dining Philosophers:**
            - Competição por recursos.
            - Pode gerar deadlock (um thread espera o outro acabar) ou livelock (thread são interrompidos continuamente por outros threads)
        - *Importante aprender estes padrões e como solucioná-los*

- Se há dois métodos synchronized na mesma classe, pode haver um bug.
    - Não usar mais de um método em um objeto compartilhado.
        - Se for necessário, requer soluções com locking
    - Synchronized deve ser evitado quando possível pois gera gargalos.
        - A seções synchronized devem ser as menores possíveis

- Shutdown correto é difícil
    - Fácil de gerar deadlock
    - Ideal é pensar nisso desde o começo.

Testing

- Testar é difícil
- Ideal é ter muitos testes e rodar com frequência
- Recomendações
    - **Não ignorar testes que falham uma vez e passam depois.**
        - Pode indicar um problema maior de threading.
    - **Começar fazendo antes funcionar a parte do código single-threaded.**
    - **Fazer código multithread ser pluggable e tunable**
        - Permite rodar de várias formas diferentes para testar.
    - **Tentar forçar erros de concurrency**
        - Rodar com mais threads que processadores
        - Rodar em plataformas diferentes com frequência.
        - Erros podem acontecer só em algumas situações bem específicas e de acaso.
            - Wait, yield e sleep e afins podem variar ordem para forçar isso
            - **Manual:**
                - Colocar esses comandos manualmente.
                - Difícil e demorado.
            - **Automatizado:**
                - Frameworks para instrumentação disso.

- Jiggling automático dos threads para alterar ordem.

14 - Sucessive Refinement

2025-10-04 8:55 am

- Estudo de Caso de um parser de argumentos de CLI
    - Mostrar etapas de desenvolvimento-
- Objetivo:
    - receber argumentos pelo CLI para o programa de uma certa forma customizada
    - `<Exemplo de bom código>`
- Código limpo primeiro é sujo e depois é limpado.
    - Primeira versão nunca é a ideal, requer trabalho repetidamente.
    - Só fazer um código que funciona nunca é o trabalho inteiro.
    - Não raro o código começa limpo e vai se bagunçando com o tempo e crescimento.
    - Existe um ponto em que é importante parar de incrementar e começar a refatorar.
    - Sempre ser incremental:
        - TDD ajuda nisso, já que sistema deve estar funcionando com testes.
        - Só progredir quando os testes passam
    - É comum na refatoração inserir código que será removido logo em seguida
        - Vários pequenos passos incrementais que vão e voltam, como resolver um cubo mágico.
    - Boa parte do bom código é compartimentalização:
        - Separar códigos diferentes em lugares diferentes
        - Separação de responsabilidades
- Muito difícil limpar código sujo há muito tempo.
    - Mantê-lo limpo desde o início vale muito mais a pena

15 - JUnit Internals

2025-10-04 09:33 am

Crítica ao framework em Java JUnit

JUnit Framework:

- Framework de testes
- Exemplo:
    - Comparar diferenças entre strings
- Como melhorar código?
    - Melhorar nomes
    - Encapsular condições do if
    - Tornar prefix argumento do findCommonSuffix para indicar acoplamento temporal
        - Mas pouco elegante, pode ainda ser melhorado.
    - Remover os +1 arbitrários com um atributo suffixLength 0-indexed.
    - Outra série de melhorias
    - Separar funções de análise e de síntese.

16 - Refactoring SerialDate

2025-10-04 09:46 am

Crítica da classe SerialDate dentro da JCommon library

Serial Date:

- Classe que representa um data em Java
  - Alternativa à date built-in do Java que é só data, sem horário.
- Testes não abrangem tudo.
  - Só metade.
  - Primeiro passo é escrever os testes para 100%
  - Consertar os testes errados e comentados
- Algumas linhas do código não são executáveis nunca.
- Retirar comentários desnecessários e excessivos.
- Melhorar o nome pouco claro:
  - Por que chama "*Serial*" Date?
  - Nome presume uma implementação, mas é uma classe abstrata.
  - Devia chamar só `Date`. Mas `DayDate` também é bom para evitar confusões.
- `MonthConstants` deve ser um `Enum`, não uma classe herdada.
  - Toda a validação de meses pode pertencer ao novo `Enum`
- Classe abstrata não pode ter detalhes de implementação
  - Definir um MAX_YEAR e MIN_YEAR não é bom
  - Abstract Factory não deve ter informações sobre seus derivados.
  - Melhor mover essa informação para a classe num nivel mais alto.
- Constantes pouco claras para que servem e de onde vieram.
- Usar keyword `final` do Java é redundante (evita alterações nas variáveis)
- Tornar funções mais expressivas
  - Usar variáveis temporárias explicativas para aumentar clareza
- Deletar funções não usadas.

17 - Smells and Heuristics

2025-10-04 11:09 am

Resumo das razões para código ruim:

Comentários:

1. **Informação inapropriada**
    - Informação que deve existir em outros lugares, como controle de versão
2. **Comentário obsoleto**
3. **Comentário redundante**
    - Não reescrever o que o código já diz por si só
4. **Comentário mal-escrito**
5. **Código comentado**

Ambiente:

1. **Build requer mais de um passo**
    - Deve ser são simples como dar um único comando
2. **Teste requer mais de um passo**
    - Também rodar testes com um único comando, de preferência direto na IDE
3. **Argumentos demais**
4. **Ter output arguments**
    - Contraintuitivo, fonte de bugs. Objeto alterado deve ser o dono do método chamado.
5. **Ter flag arguments**
    - Viola SRP, melhor separar em funções diferentes
6. **Dead Functions**
    - Código nunca chamado deve ser descartado

Geral:

1. **Múltiplas linguagens num único arquivo de código**
    - Não colocar SQL, Inglês, HTML e JavaDoc tudo junto, quanto menos melhor.
    - Ideal é uma única linguagem, mas às vezes requer mais algumas.
2. **Comportamento óbvio não é implementado**
    - *Principle of Least Surprise*
    - Funcionalidades devem ser óbvias e intuitivas.
3. **Comportamento incorreto nas *boundaries*:**
    - Inspecionar cada *edge case* e idiossincrasia.
    - Escrever testes para toda situação atípica
    - Não confiar na intuição.
4. **Segurança Burlada (overriden)**:
    - Sempre respeitar warnings, testes falhando e barreiras de segurança.
5. **Duplicação:**
    - Um dois pontos mais importantes
    - **DRY** - Don't repeat yourself
    - Duplicação é uma oportunidade de abstração.

    - Isso inclui duplicação de conceitos e responsabilidades mesmo quando o código não é igual.
6. **Código no Nível Errado de Abstração:**
    - Constantes e variáveis que são detalhe de implementação não devem existir na classe base
    - Implementações devem estar nas etapas menos abstratas do código.
7. **Classe Base dependendo das suas classes derivadas**:
    - Base classe deve ser independente de seus derivados de nível mais baixo.
    - Há exceções. E.g. código na classe base que seleciona entre as classes derivadas.
    - Devem existir em arquivos diferentes.
8. **Informação Demais:**
    - Interfaces devem ser pequenas e rasas.
    - Deve ter poucas funções.
    - Acoplamento assim fica baixo, que é bom.
    - Evitar multidão de funções utilitárias e dados privados.
9. **Dead Code**:
    - Código não executado.
    - E.g. condição que é impossível de ocorrer.
10. **Separação Vertical:**
    - Variáveis e funções devem ser definidas próximas de onde elas são usadas
11. **Inconsistência:**
    - Organizar o código sempre da mesma forma
12. **Bagunça (Clutter):**
    - Remover código sem uso ou propósito.
13. **Acoplamento Artificial:**
    - Só acoplar coisas que dependem um do outro
    - e.g. Não acoplar *enums* ou funções estáticas gerais a classes
14. **Feature Envy:**
    - Método de uma classe deve lidar apenas consigo própria e não variáveis de outros objetos.
15. **Argumentos seletores:**
    - Análogo a flag arguments, melhor dividir em várias funções.
16. **Intenção obscura:**
    - Usar nomes claros e evitar notações muito obtusas ou magic numbers
17. **Responsabilidade mal colocada:**
    - Juntar códigos que ficam intuitivamente juntos a um leitor qualquer.
18. ***Static* Inapropriado:
    - Só usar quando método realmente não depender de nada do objeto, apenas argumentos.
    - Em geral é melhor método ser não-estático
19. **Use Variáveis Explicativas:**
    - Quebrar cálculos em etapas intermediárias óbvias.
20. **Nomes devem dizer o que é feito**
21. **Entenda o Algoritmo:**
    - Não basta passar testes e parecer funcionar.
    - Devemos **saber** que está funcionando.
    - Requer refatoração com frequência.
22. **Faça Dependências Lógicas serem Físicas:**
    - Se um método depende de uma informação, isso deve estar explícito dentro daquela classe e não em outra.
23. **Prefira Polimorfismo a if/else ou switch/case**
24. **Siga Convenções Padrões:**
    - Para o time inteiro, qualquer que seja.
25. **Substitua Magic Number por Constantes Nomeadas:**

    - Podemos deixar o número mesmo para constantes muito óbvias, como o 2 em `PI * RADIUS * 2`
26. **Seja Preciso:**
    - Evitar ambiguidades e imprecisões, código deve lidar com todas as suas possibilidades, mesmo improváveis.
27. **Estrutura é mais importante que Convenção**
    - Forçar código ter uma certa estrutura é melhor do que escolher bons nomes
    - Mais rígido e previsível.
28. **Encapsule Condições:**
    - Evitar sequência longa de and e or pouco explícitos para o if/else
29. **Evite Condições Negativas**
30. **Funções devem fazer uma única coisa**
31. **Não esconda acoplamento temporal:**
    - Importante forçar uma certa ordem de execução de métodos quando esta ordem importa.
    - Colocar como argumentos aquilo que deve vir antes, mesmo que fique mais complexo.
32. **Não Seja Arbitrário:**
    - Estrutura do código deve fazer sentido e ser clara por que razão é assim.
33. **Encapsule *Boundary Conditions:***
    - Deixar tudo num lugar só, e.g. +1, -1.
34. **Funções devem descer um único nível de Abstração:**
    - Um dois pontos mais importantes e mais difíceis.
35. **Mantenha Dados Configuráveis em Alto Nível:**
    - Código de baixo nível deve receber do alto nível as constantes e variáveis necessárias.
36. **Evite Navegação Transitiva:**
    - Módulos não devem saber muito sobre outros módulos, devem ser independentes tanto quanto possível.
    - Se A depende de B e B depende de C, A não deve ter nada a ver com C.
    - *Lei de Demeter*

Java:

1. **Evite longas listas de imports, use o wildcard**
2. **Não herde constantes**
3. **Use Enums no lugar de Constantes quando apropriado**

Nomes:

1. **Escolha nomes descritivos:**
    - Requer cuidado e reescrita contínuos
2. **Escolha nomes apropriados ao nível de abstração:**
    - Não escolha nomes que comunicam implementação.
    - E.g. misturar *Modem* com *phoneNumber*, níveis diferentes
3. **Use nomenclatura padrão quando possível:**
    - Usar o nome *Decorator* para um código no padrão Decorator.
4. **Não use nomes ambíguos:**
    - Ser bem claro mesmo que o nome seja longo
5. **Use Nomes Longos para escopos maiores**
6. **Evite Encodings**:
    - e.g. prefixos ou notação húngara
7. **Nomes devem descrever side-effects**

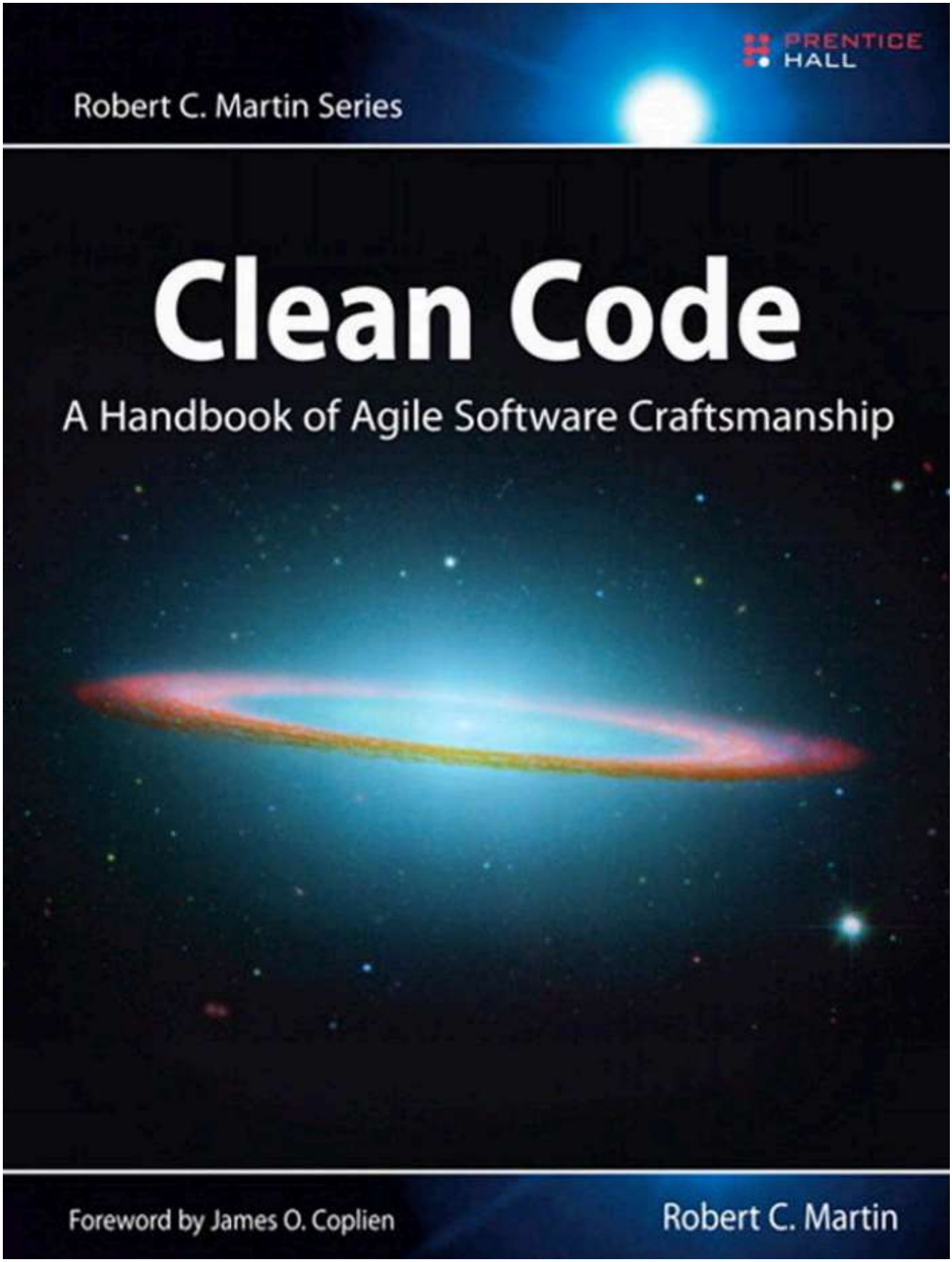
Testes:

1. **Testes Insuficientes**
2. **Use uma ferramenta de cobertura**
3. **Não pule testes triviais**
4. **Um teste ignorado pode gerar ambiguidade de comportamento**
5. **Teste *Boundary Conditions***
6. **Concentre mais testes onde há mais bugs**
    - Bugs costumam andar em bando
7. **Padrões de falha de testes podem revelar razões de problema no código**
8. **Padrões de Cobertura de testes podem revelar problemas**
9. **Testes devem ser rápidos**

_README

2025-10-02 18:35 pm

Read PDF'


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

README1

- book_title: Clean Code
- subtitle: A Handbook of Agile Software Craftsmanship
- authors:
  - Robert C. Martin
- publisher: Prentice Hall
- date_created: 2025-10-02T18:35:00-03:00
- tags:
  - #book
  - #software
- pages: 462
- started: true
- finished: true