# Linguagens de Programação: Orientação a Objetos Boas Práticas

Prof. Arnaldo Candido Junior

UNESP – IBILCE

São José do Rio Preto, SP

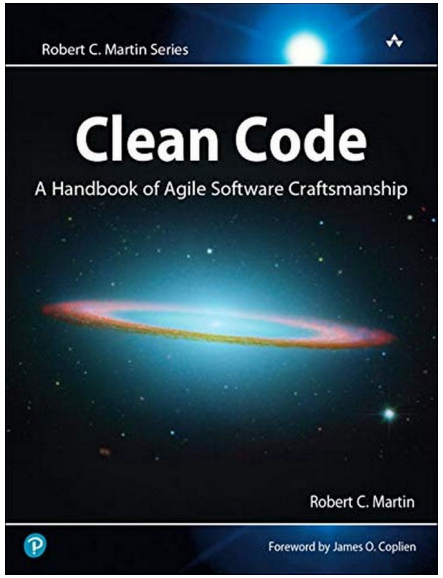
# Introdução

- Nos últimos anos, houve uma explosão de linguagens, bibliotecas e frameworks
- As técnicas de programação e boas práticas amadureceram
  - Exemplos: Clean code, SOLID, Low Coupling
- Dominar as técnicas requerem experiência. Este material traz um roteiro sobre as principais

# Código básico

- Evite usar variáveis globais
- Seja consistente com a formatação do código
  - Ex.: identação, nomes de variáveis
- Seja consistente com a estratégia de codificação
  - Ex.: busque usar as mesmas estruturas de dados quando possível

# Clean Code




- Livro Robert C Martin
- Pontos chaves a seguir
- Resumo do resumo de: https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

# Clean Code (2)

- Siga convenções: identação, case (camel case, sneak case, …), entre outros
- Aplique Filosofia Kiss: busque manter a lógica simples
- Use a regra do escoteiro: quando trabalhar sobre um trecho de código, deixe-o mais organizado do que quando você chegou
- Busque a causa principal do problema ao analisá-lo

# Clean Code (3)

- Evite o excesso de configurações
- Seja consistente na forma de codificar
- Use injeção de dependências (facilita os testes)
- Use nomes descritivos e inequívoco
  - De preferência, explicativas (o nome indica o que está acontecendo no código)

# Clean Code (4)

- Funções / métodos
  - Use nomes descritivos para funções
  - Construa funções pequenas
  - Projete funções que façam uma coisa, fazendo essa coisa bem-feita
  - Prefira funções com poucos argumentos
  - Prefira funções sem efeitos colaterais

# Clean Code (5)

- Comentários
  - Busque explicar a intenção por trás do código
  - Evite comentários redundantes ou óbvios
  - Remova código sem uso em vez de deixá-lo comentado
  - Use-os para clarificar funcionamento do código quando necessário
  - Deixe avisos sobre consequências daquele código

# Clean Code (6)

- Deixe juntas as funções dependentes e as similares
  - Como matérias relacionadas ficam juntas em um jornal
- Declare as variáveis próximo de onde for usá-las
- Deixe as linhas de código curtas

# Clean Code (7)

- Code smells (sintomas de problemas)
  - Código muito rígido (difícil de mudar)
  - Código frágil (quebra após uma mudança simples)
  - Código imóvel (difícil de reusar sem grandes alterações)
  - Complexidade desnecessária / repetição desnecessária
  - Código difícil de entender

# SOLID

- **Single Responsability**: uma classe deve ter um único trabalho a fazer e fazê-lo bem feito
- **Open-closed principle**: classes devem ser abertas a extensões e fechadas a modificações
- **Liskov substitution**: código que usa uma classe básica deve funcionar com classes derivadas sem modifcações

# SOLID (2)

- **Interface segregation**: clientes não devem ser forçados a depender de interfaces que não usam
- **Inversão de dependência**: clientes devem depender de interfaces, não de classes
  - Simplifica testes

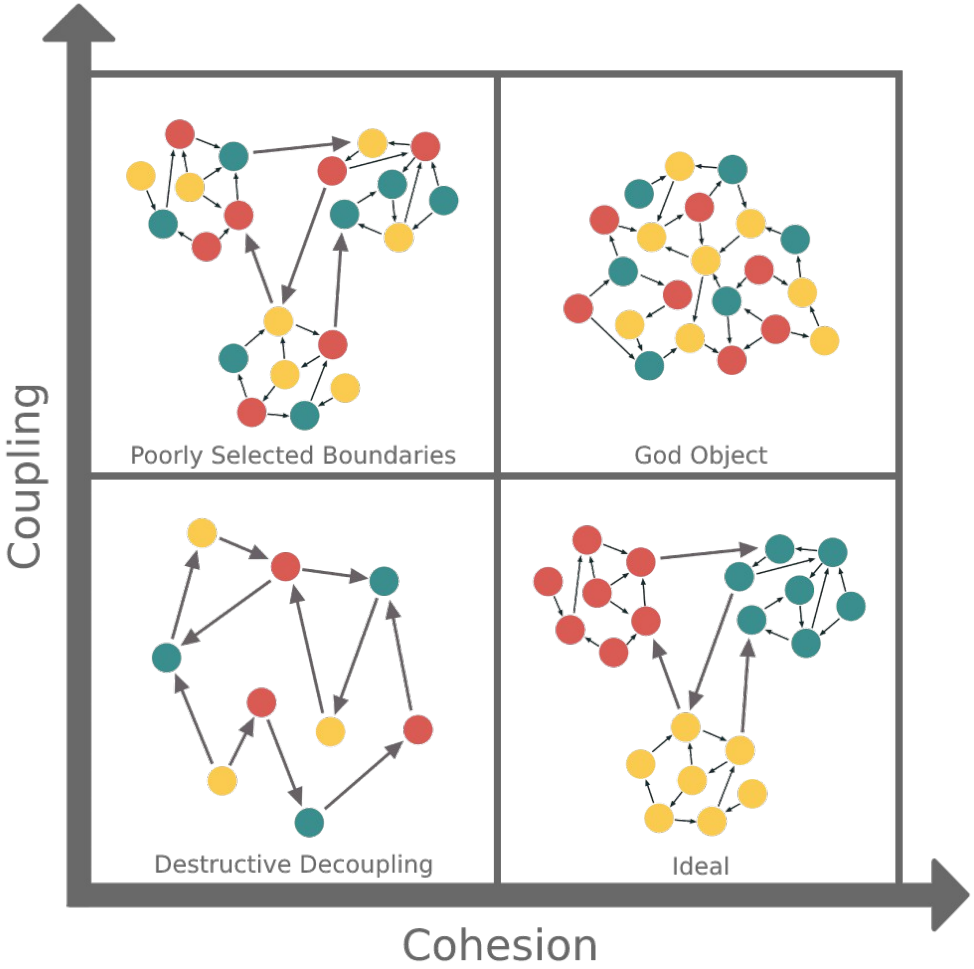
# Baixo Acoplamento (2)

- **Low coupling, high coesion**: permite o reuso do código
- **Baixo acoplamento**: um módulo depende o mínimo possível dos outros módulos do sistema
- **Alta coesão**: os componentes de um módulo tem o mesmo propósito, dependem fortemente uns dos outros

# Baixo Acoplamento (2)

- The Pragmatic Programmer:
  - Desejamos construir componentes autocontidos, independentes e com um propósito simples e bem definido
- **Opinião**: quando conflitar com DRY, preferir baixo acoplamento

# Baixo Acoplamento (2)

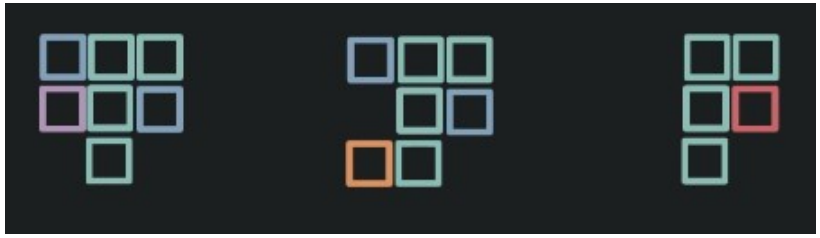


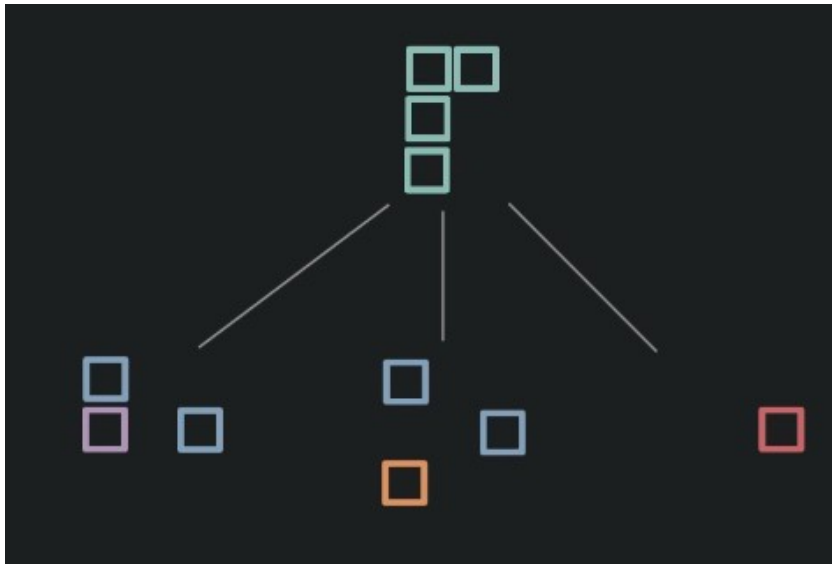

Fonte: https://feature-sliced.design/docs/reference/isolation/coupling-cohesion

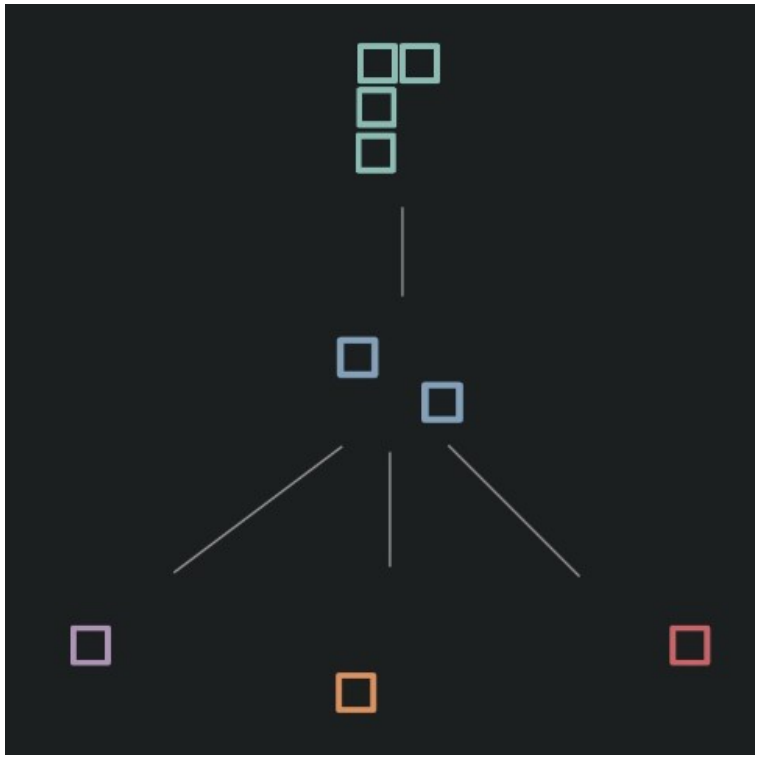
# Composition over Inheritance

- Herança: vista como muito poderosa quando foi concebida
- Com o passar o tempo, mostrou-se pouco flexível em diferentes cenários
- Principal problema: aumenta o acoplamento
- Quando se herda uma classe, herda-se tudo de todos os ancestrais, até o que não é desejado
  - Mudanças em uma classe ancestral tem um efeito cascata muito forte sobre os decendentes

# Composition over Inheritance (2)





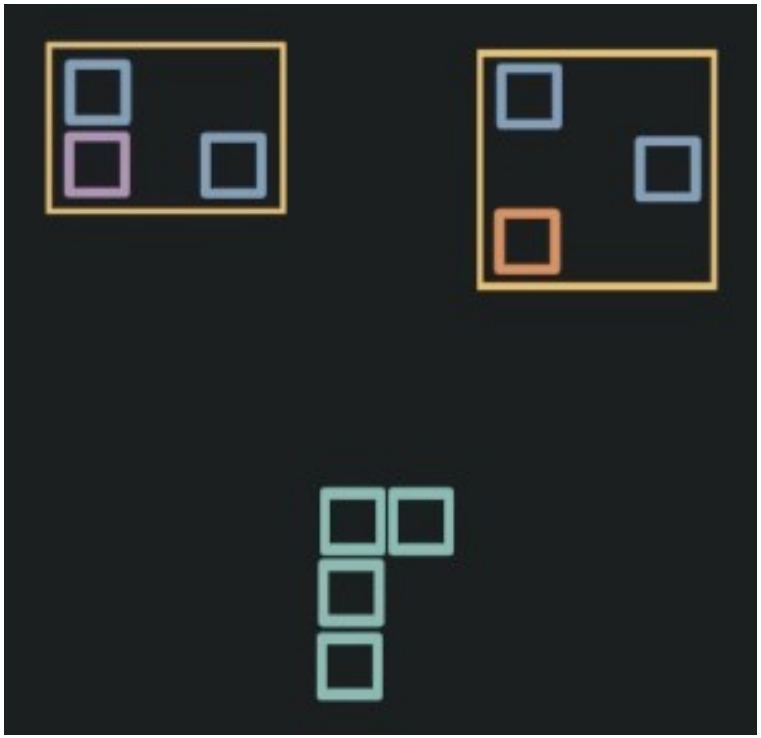


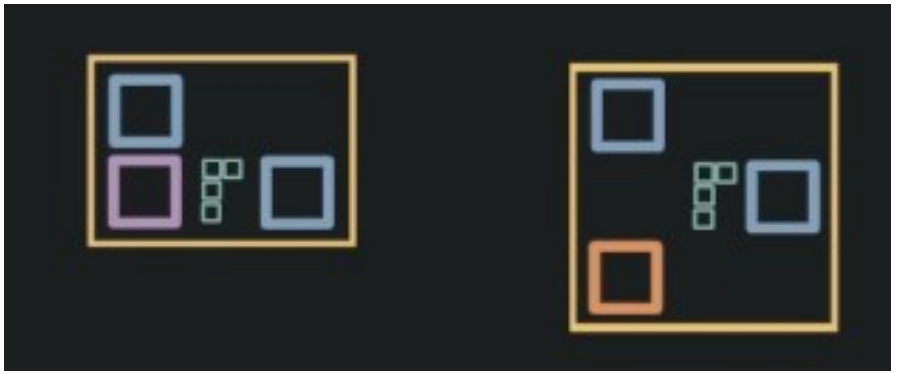
Fonte: https://www.youtube.com/watch?v=hxGOiiR9ZKg

# Composition over Inheritance (3)

- Principio: preferir composição sobre herança
- Intuição: como montar um lego. Classes usam apenas aquilo que realmente vão precisar
- Importante: ainda existem casos em que herança faz sentido
- Aprofundamento da discussão:
  https://www.youtube.com/watch?v=hxGOiiR9ZKg

# Composition over Inheritance (4)





Fonte: https://www.youtube.com/watch?v=hxGOiiR9ZKg

# Princípios famosos

- **Do not Repeat Yourself (DRY)**: evite duplicação de código quando possível
- **Convention Over Configuration (CoC)**: é melhor seguir convenções do nicho que personalizar o projeto
  - Ex.: localização de arquivos de código e arquivos HTML

# Outros

- Vale a pena ficar de olho em:
  - **Arquiteturas** de desenvolvimento. Ex.: MVC (Model-View-Control) para o nicho Web
  - **Metodologias ágeis** de desenvolvimento: Scrum, XP, entre outras
  - **Padrões de projeto**: soluções em linguagem com tipagem forte para alguns problemas comuns

# Limitações

- Boas práticas não são regras absolutas. Se fossem seriam reforçadas pelo compilador...
- Tendem a focar programação comercial
- Podem parecer burocráticas. Mais simplificam muito a criação de software complexos

# Limitações (2)

- Podem ser vagas. Exemplo: o que é fazer uma única coisa em Single Responsatiblity?
  - Ficam mais claras com a **experiência**
- Podem ser conflitantes. Estão sujeitas a trade-offs, como tudo na computação está

# Limitações (3)

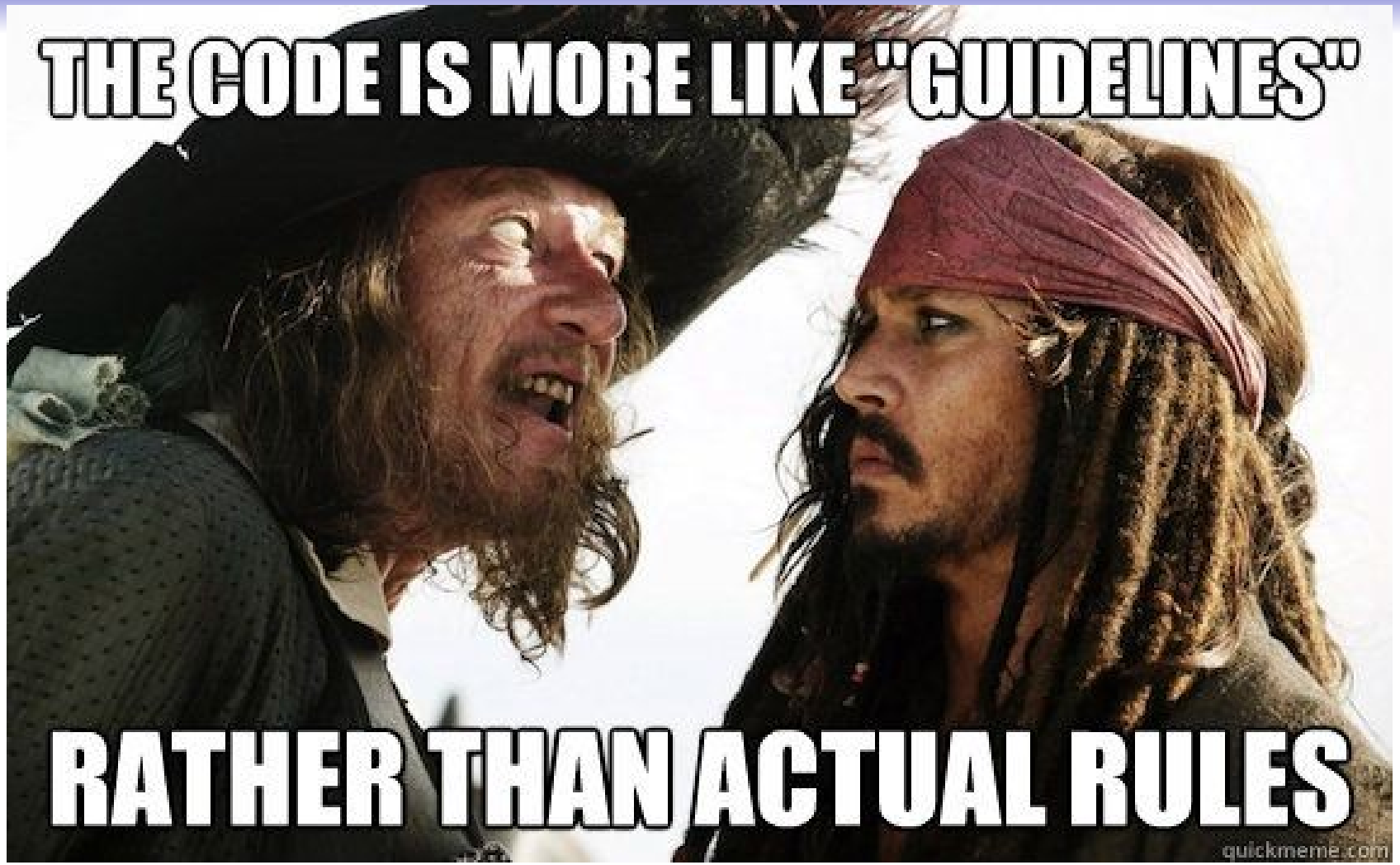
- As boas práticas apresentadas não são universais, principalmente fora do eixo de programação comercial
- Podem ser desnecessárias em scripts simples (redes e servidores; ciência de dados)
- Podem ficar em segundo plano mesmo em softwares de grande porte. Ex.: foco em “escovação de bits” no kernel do Linux

# Opinião pessoal

- **Desempenho** vs **legibilidade**: focar em legibilidade
  - É mais barato comprar hardware extra do que aumentar a equipe de desenvolvedores
  - Livro texto: manutenção custa mais que desenvolvimento geralmente
  - Otimização precoce: quando tempo gasto com otimização gera pouco aumento no desempenho

# Opinião pessoal (2)

- **Tipagem**: preferir tipagem estática e forte
  - Aumenta o tempo para desenvolver, mas diminui o tempo para depurar
  - Permite capturar erros precocemente, antes que eles ocorram na produção
  - Foi o que motivou o surgimento de linguagens como Typescript


THE CODE IS MORE LIKE "GUIDELINES"
RATHER THAN ACTUAL RULES
quickmeme.com

# Considerações Finais

- Foque-se na criação de códigos que:
  - Sejam fáceis de se entender
  - Sejam simples de modificar, pelo ajuste as funcionalidades atuais
  - Sejam flexíveis para se estender, pela adição de novas funcionalidades
- Programar é mais uma **arte** do que uma **ciência**