Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Prof. Jackson Barreto

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parceiros para o Crescimento
PORTUGAL DIGITAL

Objetivo

Compreender e aplicar os princípios de *Clean Code* (Código Limpo) e os princípios SOLID no desenvolvimento de software em JavaScript, com o intuito de melhorar a legibilidade, manutenção e escalabilidade do código.

Objetivos Específicos

Ao final da sessão, os formandos deverão ser capazes de:

- Identificar "Bad Smells" comuns no código;
- Explicar os cinco princípios SOLID (S, O, L, I, D) e a sua importância para a qualidade do design de software;
- Executar técnicas de *Refactoring* em código existente para melhorar a sua estrutura sem alterar o comportamento funcional.

Innominatum CONSULTORIA

Clean Code

A Arte de Escrever Código Legível – Robert C. Martin (“Uncle Bob”)

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

O que é Código Limpo?

😰 **Cozinha Bagunçada**

- Utensílios espalhados
- Ninguém encontra nada
- Leva tempo para cozinhar
- Fácil contaminar comida

😊 **Cozinha Organizada**

- Tudo no seu lugar
- Qualquer um encontra
- Rápido e eficiente
- Seguro e higiénico

Código é igual a uma cozinha: outros vão usar depois de si!

Bad Smells

Bad Smells são sinais de que alto está errado no código. Não é um erro, mas indica problemas de design.

Vamos treinar o seu "Nariz" do Programador 👃

Bad Smells

Vamos identificar 4 *Bad Smells* clássicos.

- 📏 Funções Gigantes
- ❓ Nomes Enigmáticos
- 🌀 If/Else Aninhados
- 🔢 Números Mágicos

Clean Code: Funções Pequenas

Funções Gigantes = Bed Smell

Sintomas:

- Precisa de scroll para ver tudo
- Difícil de explicar o que faz
- Muitos comentários a separar secções

Clean Code: Funções Pequenas

Uma função deve fazer apenas UMA coisa, e fazê-la bem.

❌ **Função que faz tudo**

```
function processarDados(dados) {
  // valida
  // transforma
  // calcula
  // guarda
  // envia email
  // gera relatório
  // ... 100 linhas
}
```

✅ **Funções especializadas**

```
function processarDados(dados) {
  const validados = validar(dados);
  const transformados = transformar(validados);
  const resultado = calcular(transformados);
  guardar(resultado);
  notificar(resultado);
}
// Cada função: 5-15 linhas
```

Benefícios: Fácil de testar, reutilizar, e manter!

Clean Code: Convenções de Nomeclatura

Nomes Enigmáticos = Bed Smell

💡 **Pergunta-te: Daqui a 6 meses, vou perceber este nome?**

Se a resposta é "talvez não", muda o nome agora!

Clean Code: Convenções de Nomeclatura

Variáveis	`camelCase`	`nomeCompleto, idadeUtilizador`
Constantes	`UPPER_SNAKE`	`MAX_TENTATIVAS, API_URL`
Funções	`camelCase + verbo`	`calcularTotal(), obterUser()`
Classes	`PascalCase`	`GestorEncomendas, Utilizador`
Booleanos	`is/has/can + nome`	`isAtivo, hasPermissao`

O nome deve revelar a intenção, sem precisar de comentário!

Clean Code: Convenções de Nomeclatura

❌ O que faz este código?

```
function calc(l, u) {
  if (u.a) {
    return l.filter(i => i.s === 1);
  }
  return l;
}
```

✅ Agora percebe-se!

```
function filtrarUtilizadores(lista, filtro) {
  if (filtro.apenasAtivos) {
    return lista.filter(
      user => user.estado === ATIVO
    );
  }
  return lista;
}
```

Clean Code: Cláusula de Guarda

If/Else Aninhados = Bed Smell

```
if (user) {
  if (user.isActive) {
    if (user.hasPermission) {
      if (product.inStock) {
        if (payment.isValid) {
          // finalmente! 😵
        }
      }
    }
  }
}
```

Clean Code: Cláusula de Guarda

✅ **Early Return**

```
if (!user) return;
if (!user.isActive) return;
if (!user.hasPermission) return;
if (!product.inStock) return;
if (!payment.isValid) return;

// Código principal aqui ✨
processarCompra();
```

Se você já sabe que a função não pode continuar, **saia dela imediatamente**.

Clean Code: Constante, Intenção e Verdade

Números Mágicos = Bed Smell

❌ O que significa 1? E 86400?

```
if (user.status === 1) {
  // ativo? inativo? pendente?
}

const timeout = 86400;
// 86400 o quê? segundos? ms?
```

✅ Usar constantes com nomes

```
const STATUS_ATIVO = 1;
const UM_DIA_EM_SEGUNDOS = 86400;

if (user.status === STATUS_ATIVO) {
  // Agora é claro!
}
const timeout = UM_DIA_EM_SEGUNDOS;
```

Clean Code: Constante, Intenção e Verdade

- Constantes Nomeadas (Named Constants)
- Intenção Reveladora (Revealing Intent)
- Single Source of Truth (Fonte Única de Verdade)

```
const STATUS_ATIVO = 1;
const UM_DIA_EM_SEGUNDOS = 86400;
```

Clean Code: DRY

❌ **Código repetido**

```
// Em 5 sítios diferentes...
const preco1 = quantidade * precoUnit;
const iva1 = preco1 * 0.23;
const total1 = preco1 + iva1;

const preco2 = quantidade * precoUnit;
const iva2 = preco2 * 0.23;
const total2 = preco2 + iva2;
// Se o IVA mudar? 😰
```

✅ **Uma única fonte de verdade**

```
const TAXA_IVA = 0.23;

function calcularTotal(qtd, precoUnit) {
  const subtotal = qtd * precoUnit;
  const iva = subtotal * TAXA_IVA;
  return subtotal + iva;
}

// Usar em qualquer lado:
const total1 = calcularTotal(5, 10);
const total2 = calcularTotal(3, 25);
```

Se copiaste e colaste código, provavelmente precisas de uma função!

SOLID

SOLID: O Acrónimo

S — **Single Responsibility**
Uma classe, uma responsabilidade

O — **Open/Closed**
Aberto para extensão, fechado para modificação

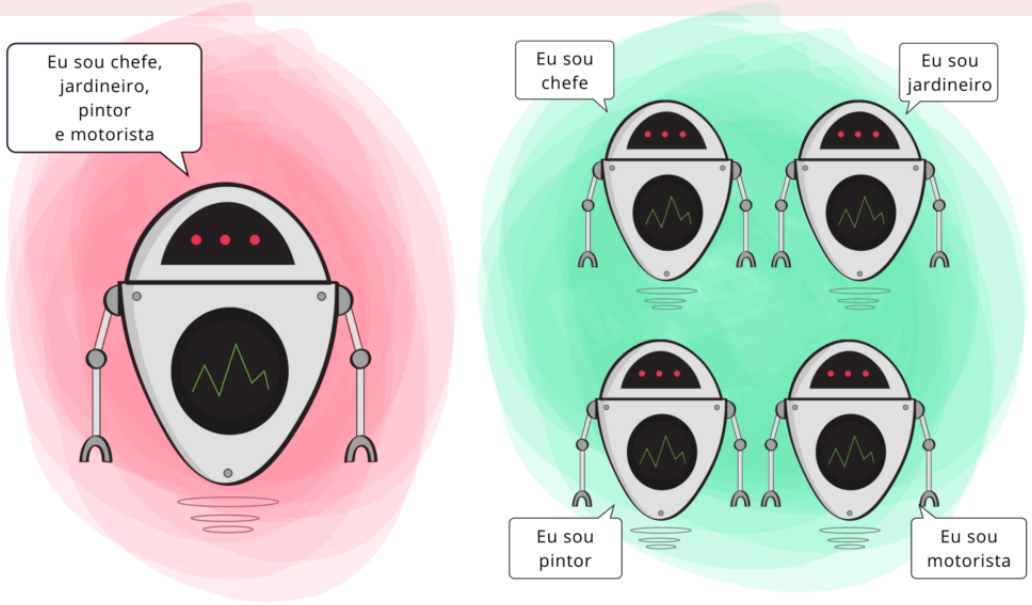
L — **Liskov Substitution**
Subclasses substituem a classe pai

I — **Interface Segregation**
Interfaces específicas são melhores

D — **Dependency Inversion**
Depender de abstrações, não de concretos

Single Responsability Principle (SRP)

Uma classe deve ter apenas UMA razão para mudar.



Responsabilidade Única

Cada um é especialista e pode ser substituído sem afetar os outros!

Fonte: https://warcontent.com/principios-solid-imagens/

Single Responsability Principle (SRP)

❌ Classe "OrderManager"

```
class OrderManager {
  calcularTotal() { ... }
  validarStock() { ... }
  enviarEmail() { ... }
  gerarPDF() { ... }
}

// 4 razões para mudar!
// Viola SRP
```

✅ Classes especializadas

```
class CalculadoraPrecos {
  calcularTotal() { ... }
}
class GestorStock {
  validarStock() { ... }
}
class ServicoEmail {
  enviarEmail() { ... }
}
class GeradorPDF {
  gerarPDF() { ... }
}
```

Se mudar a lógica de email, tenho de alterar OrderManager?

Se sim → viola SRP. Se cada mudança afeta apenas uma classe → cumpre SRP!

Open/Closed Principle

Aberto - Fechado

Fonte: https://warcontent.com/principios-solid-imagens/

Open/Closed Principle

Aberto para extensão, fechado para modificação.

🎫 Exemplo: Sistema de Tickets

❌ Modificar para cada tipo

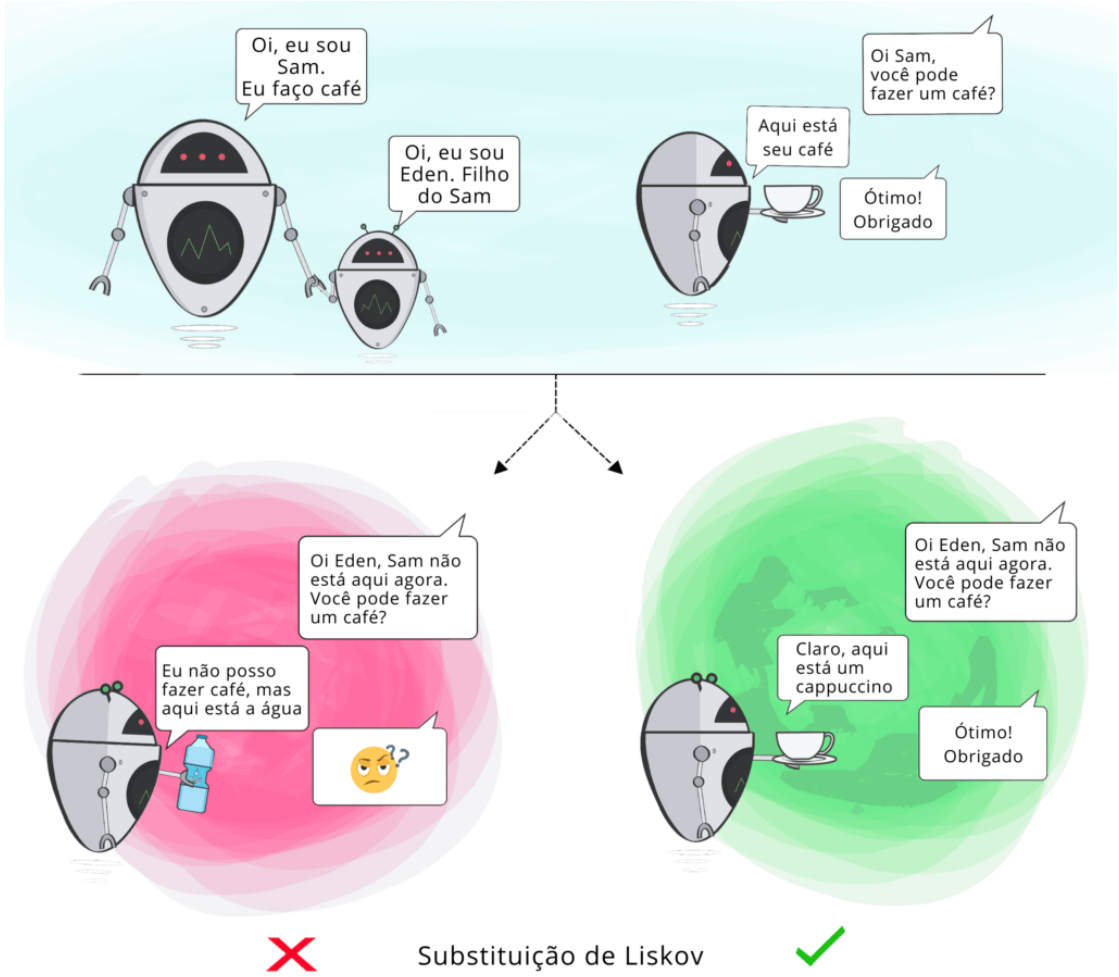
```
function calcularPrioridade(ticket) {
  if (ticket.tipo === "bug") {
    return ticket.severidade * 2;
  } else if (ticket.tipo === "feature") {
    return ticket.impacto;
  }
  // Novo tipo? Alterar aqui...
}
```

✅ Estender sem modificar

```
// Cada tipo sabe calcular
const calculadores = {
  bug: (t) => t.severidade * 2,
  feature: (t) => t.impacto,
  // Adicionar novo: só criar
  hotfix: (t) => t.severidade * 3
};

const prioridade = calculadores[tipo](ticket);
```

Adicionar funcionalidade = criar código novo, não alterar código existente

Liskov Substitution Principle



Se S é um subtipo de T, então objetos do tipo T em um programa **podem ser substituídos** por objetos do tipo S sem alterar nenhuma das propriedades desse programa.

Fonte: https://warcontent.com/principios-solid-imagens/

Liskov Substitution Principle

❌ Viola LSP

```
class Ave {
  voar() {
    return "A voar...";
  }
}

class Pinguim extends Ave {
  voar() {
    throw new Error("Não voo!");
  }
}

// Problema: código que espera Ave
// vai falhar com Pinguim!
function fazerVoar(ave) {
  return ave.voar(); // 💥 Erro!
}
```

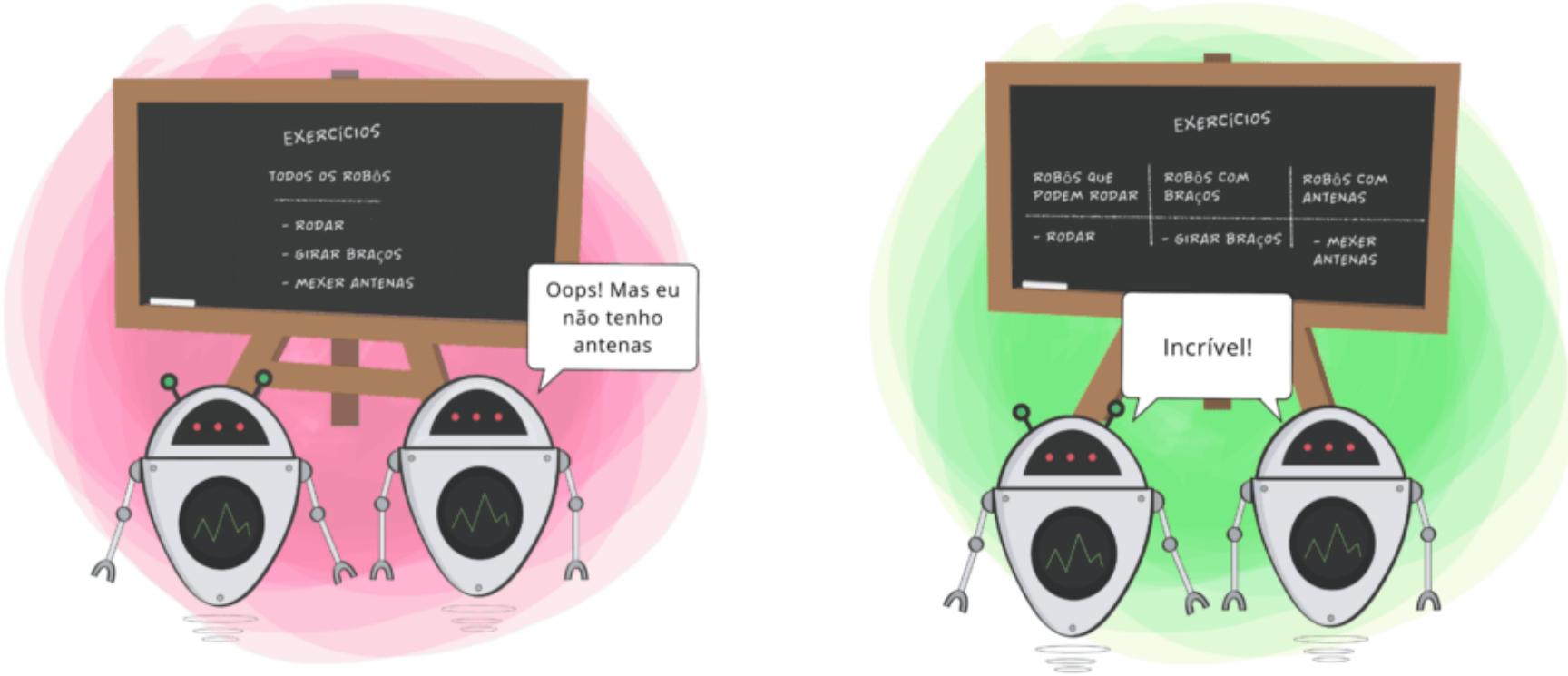
✅ Respeita LSP

```
class Ave {
  mover() { return "A mover..."; }
}

class AveVoadora extends Ave {
  voar() { return "A voar..."; }
}

class Pinguim extends Ave {
  nadar() { return "A nadar..."; }
}

// Agora cada subtipo cumpre
// o contrato da sua classe pai
const aguia = new AveVoadora();
const pingu = new Pinguim();
```

Regra: Se uma função espera a classe pai, qualquer filho deve funcionar.

Subclasses não devem "surpreender" removendo funcionalidades esperadas.

Interface Segregation Principle



Fonte: https://warcontent.com/principios-solid-imagens/

✗ Segregação de Interface ✓

Uma classe não devem ser forçada a implementar métodos que não utiliza.

Interface Segregation Principle

❌ Interface "gorda"

```
// Interface com tudo
const Trabalhador = {
  trabalhar: () => {},
  comer: () => {},
  dormir: () => {}
};

// Robot não come nem dorme!
const robot = {
  trabalhar: () => "A trabalhar",
  comer: () => { /* não faz sentido */ },
  dormir: () => { /* não faz sentido */ }
};

// Robot é forçado a implementar
// métodos que não usa 😳
```

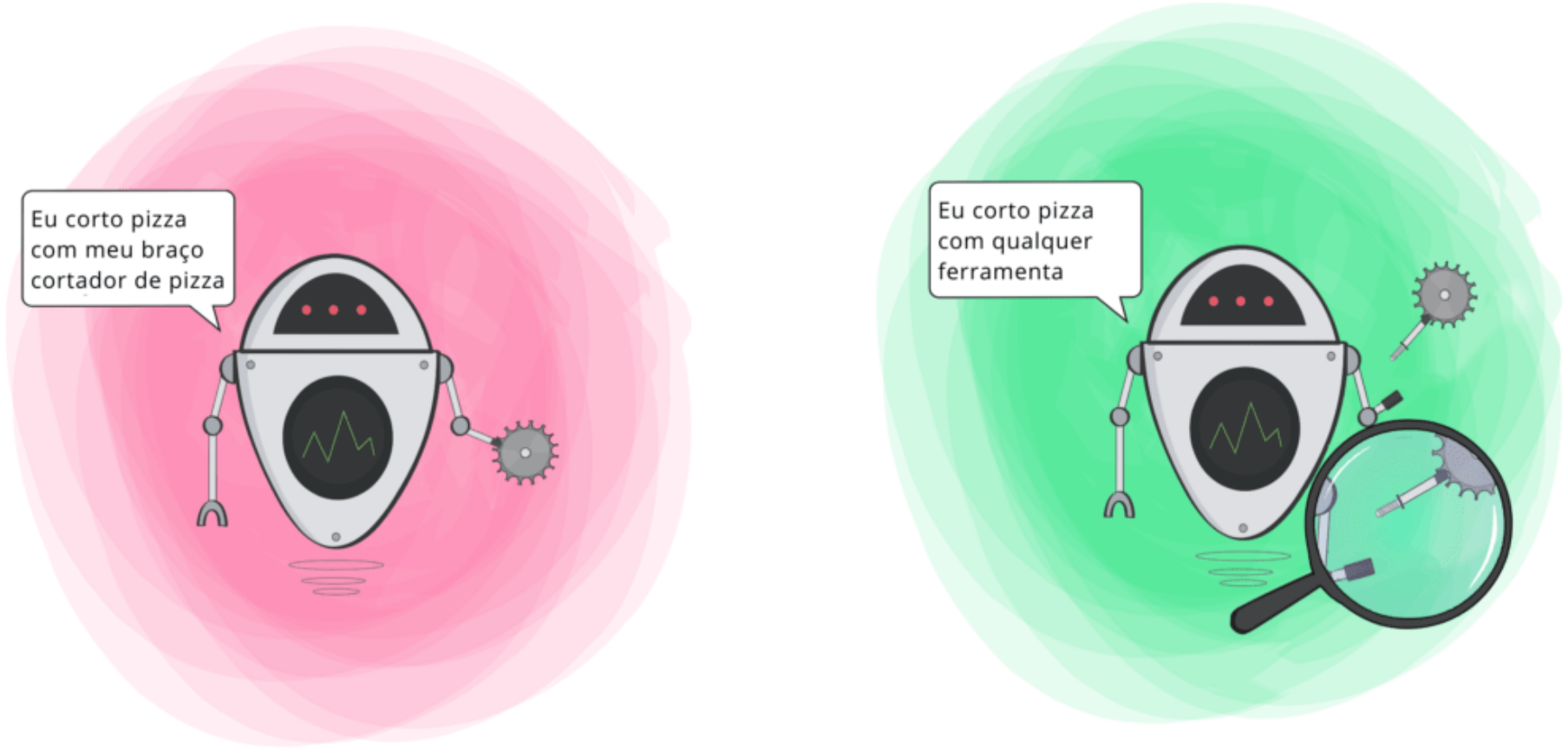
✅ Interfaces segregadas

```
// Interfaces pequenas e específicas
const Trabalhavel = {
  trabalhar: () => {}
};

const Alimentavel = {
  comer: () => {}
};

// Humano usa as duas
const humano = {
  ...Trabalhavel,
  ...Alimentavel,
  trabalhar: () => "A trabalhar",
  comer: () => "A comer"
};

// Robot só usa o que precisa
const robot = {
  ...Trabalhavel,
```

Regra: Ninguém deve ser forçado a depender de métodos que não usa.

Prefira várias interfaces pequenas a uma interface gigante.

Dependency Inversion Principle



Fonte: https://warcontent.com/principios-solid-imagens/

✗ Inversão de Dependência ✓

Dependency Inversion Principle

Depender de abstrações, não de implementações concretas.

🔌 Analogia: Tomadas elétricas

❌ Ligação directa

Fios soldados direto à parede
Mudar TV = chamar eletricista
Dependência rígida

✅ Usar tomada (abstração)

Qualquer aparelho funciona
Trocar TV = ligar na tomada
Flexibilidade total

Módulos de alto nível não devem depender de módulos de baixo nível

Dependency Inversion Principle

❌ Dependência direta

```
class MySQLDatabase {
  guardar(dados) {
    // código específico MySQL
  }
}

class GestorUtilizadores {
  constructor() {
    // Dependência DIRETA de MySQL
    this.db = new MySQLDatabase();
  }

  criarUser(user) {
    this.db.guardar(user);
  }
}

// Problema: mudar para MongoDB?
```

✅ Depende de abstração

```
// "Contrato" abstrato
const Database = {
  guardar: (dados) => {}
};

class GestorUtilizadores {
  constructor(database) {
    // Recebe qualquer database!
    this.db = database;
  }

  criarUser(user) {
    this.db.guardar(user);
  }
}

// Fácil trocar implementação:
const gestor1 = new GestorUtilizadores(mysql);
```

Regra: Módulos de alto nível não devem depender de módulos de baixo nível.

Ambos devem depender de abstrações (injeção de dependências).

SOLID: Resumo

S

Uma coisa

O

Estender, não modificar

L

Substituível

I

Só o necessário

D

Abstrações

"Código que segue SOLID é mais fácil de testar, manter e evoluir!"

Refactoring

"Melhorar código sem mudar comportamento"

🔧 **Refactoring = "Limpar a casa"**

O código continua a fazer exatamente o mesmo, mas fica mais limpo, organizado e fácil de entender.

🕐 Quando fazer?

- Antes de adicionar funcionalidade nova
- Quando encontra um *bad smells*
- Durante Code Review

Técnicas de Refactoring

Renomear	Mudar nomes para serem mais claros
Extrair Função	Transformar bloco de código em função
Extrair Variável	Dar nome a expressões complexas
Remover Duplicação	Aplicar DRY
Simplificar Condições	Early return, guard clauses

⚠️ **Sempre testar depois de refatorar para garantir que funciona igual!**

Refactoring: Antes vs Depois

❌ ANTES

```
function p(l, u) {
  let t = 0;
  for (let i = 0; i < l.length; i++) {
    if (l[i].s === 1) {
      t = t + l[i].p * l[i].q;
      if (u.d) {
        t = t - (t * 0.1);
      }
    }
  }
  t = t + (t * 0.23);
  return t;
}
```

✅ DEPOIS

```
const ATIVO = 1;
const DESCONTO = 0.1;
const IVA = 0.23;

function calcularTotal(produtos, user) {
  const ativos = filtrarAtivos(produtos);
  let subtotal = somarPrecos(ativos);

  if (user.temDesconto) {
    subtotal = aplicarDesconto(subtotal);
  }

  return aplicarIVA(subtotal);
}
```

Mudanças: Nomes claros | Constantes | Funções pequenas | Sem números mágicos

O comportamento é o mesmo, mas agora qualquer pessoa percebe!

Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Prof. Jackson Barreto

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parceiros para o Crescimento
PORTUGAL DIGITAL