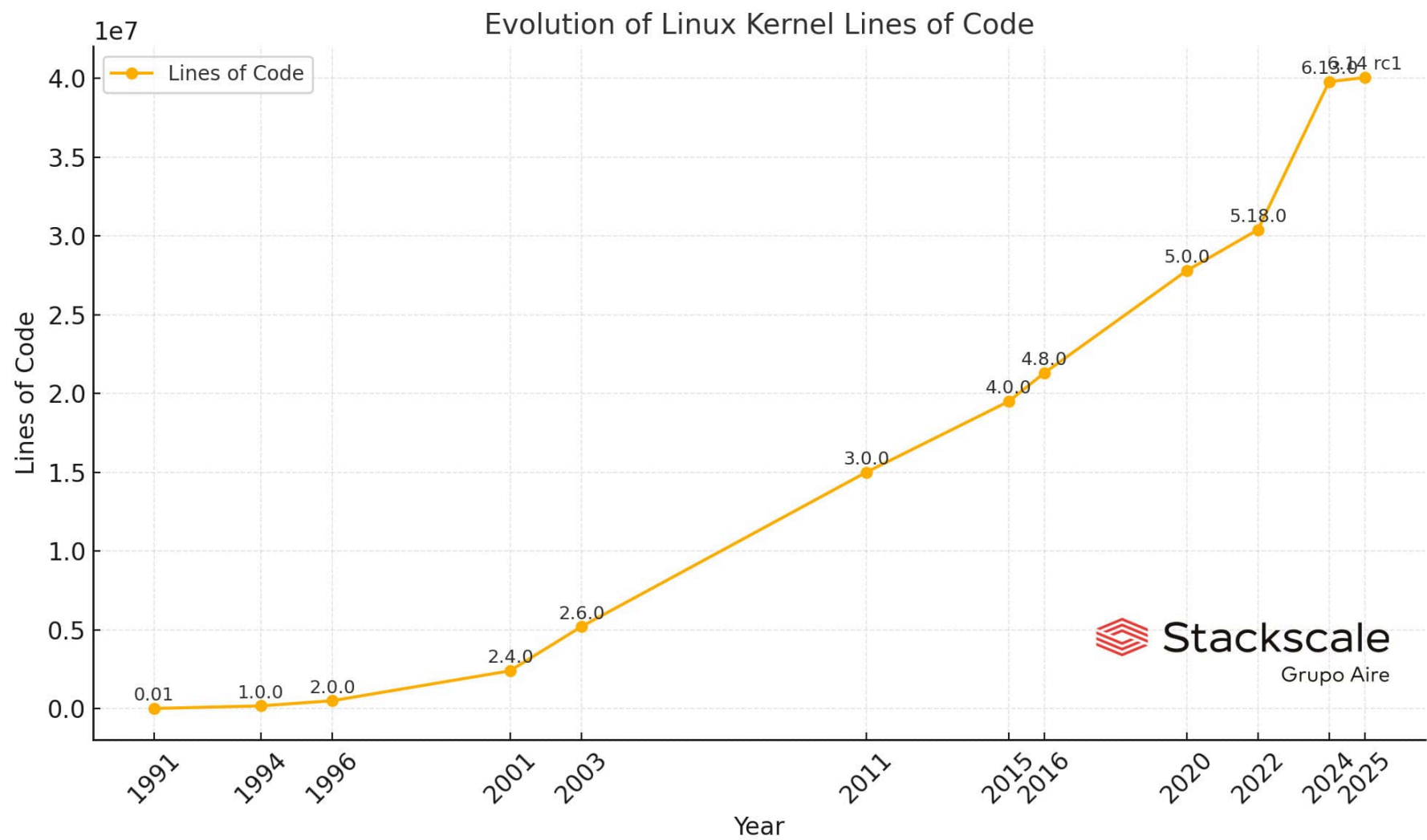


TÓPICO 02: PRINCÍPIOS SOLID

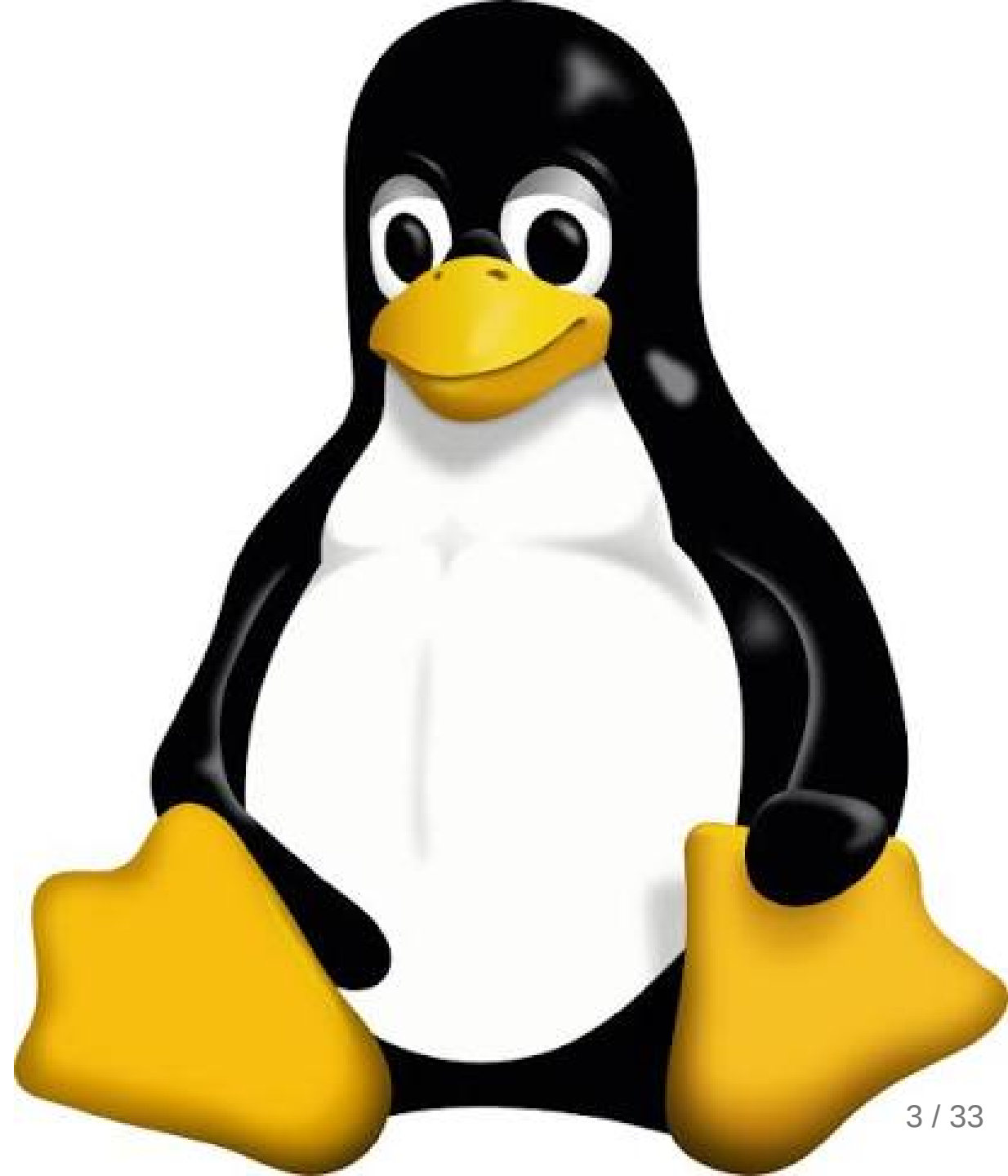
Design Patterns - Professor Ramon Venson - SATC 2026.2



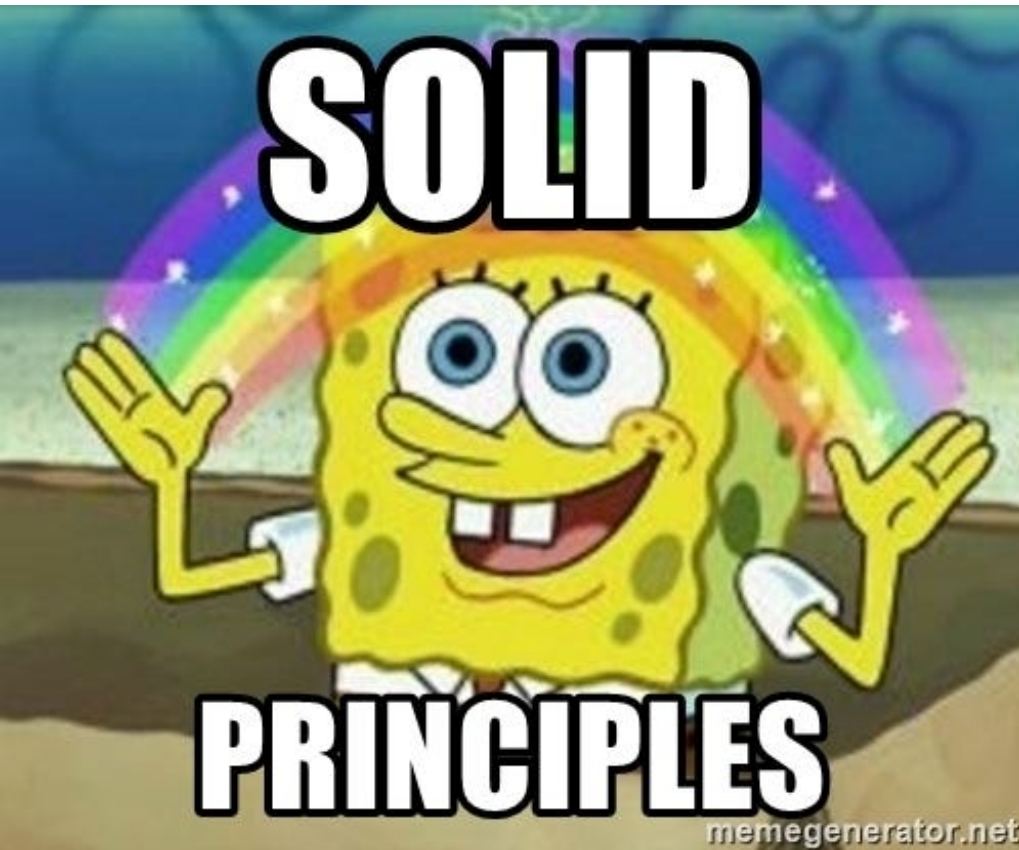
Evolução de Software

O código fonte do Linux Kernel tem mais de **40 milhões de linhas**, mas começou com apenas **10 mil linhas** em 1991.

Restam aproximadamente **250 linhas do código original** que permanecem intocadas (~2.5%)



Como escrever código que
continua funcionando depois de
dezenas, centenas ou milhares de
alterações?



SOLID

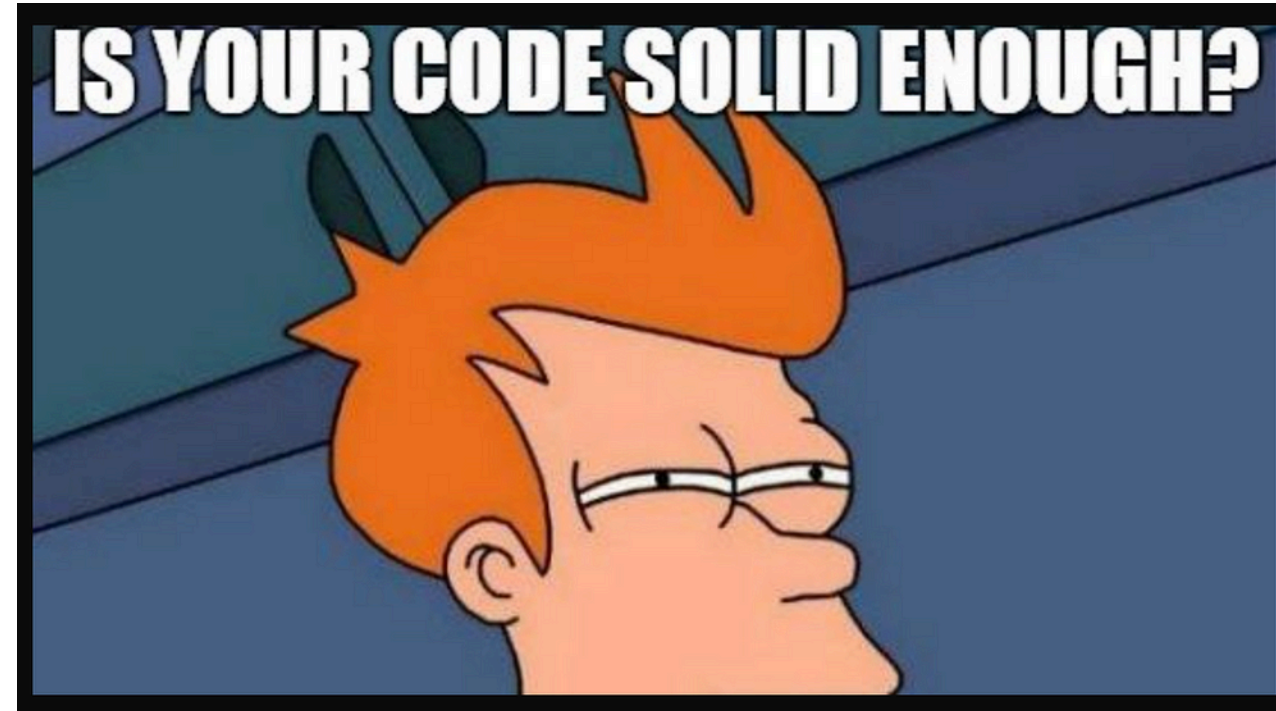
SOLID é um acrônimo para **5 princípios** de design de software orientado a objetos, que ajudam a criar sistemas mais **flexíveis**, **modulares** e **fáceis de manter**.

É mais como um **guia** do que uma regra rígida, mas é amplamente aceito como boas práticas de design.

Conjunto de **5 princípios** para OOP:

- **S** – Single Responsibility
- **O** – Open/Closed
- **L** – Liskov Substitution
- **I** – Interface Segregation
- **D** – Dependency Inversion

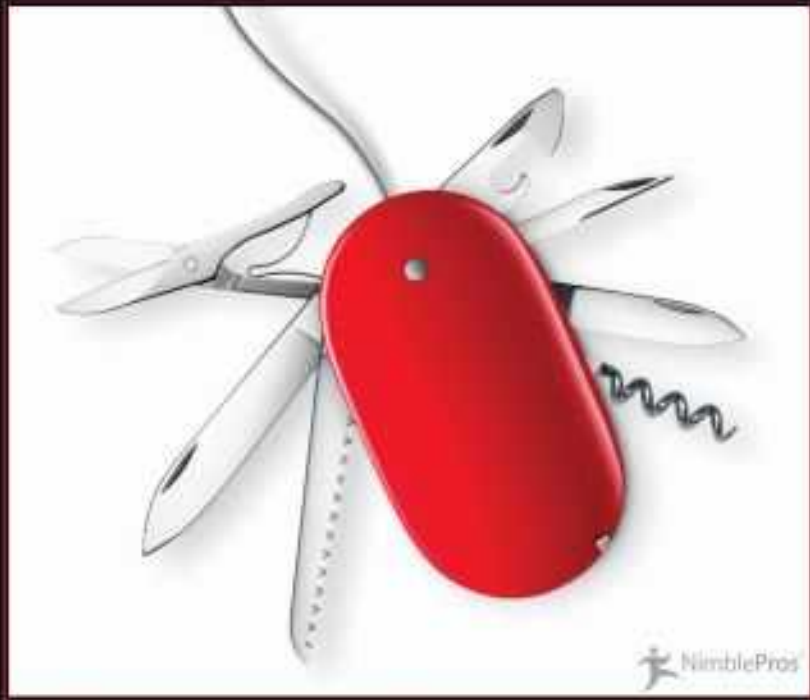
Objetivo: código **flexível**, **modular** e **fácil de manter**.



S – Single Responsibility

Uma classe deve ter **apenas uma razão para mudar**.

- Uma responsabilidade principal
- Aumenta coesão
- Facilita testes e manutenção



SINGLE RESPONSIBILITY

Avoid tightly coupling your tools together.

SRP – Mau exemplo

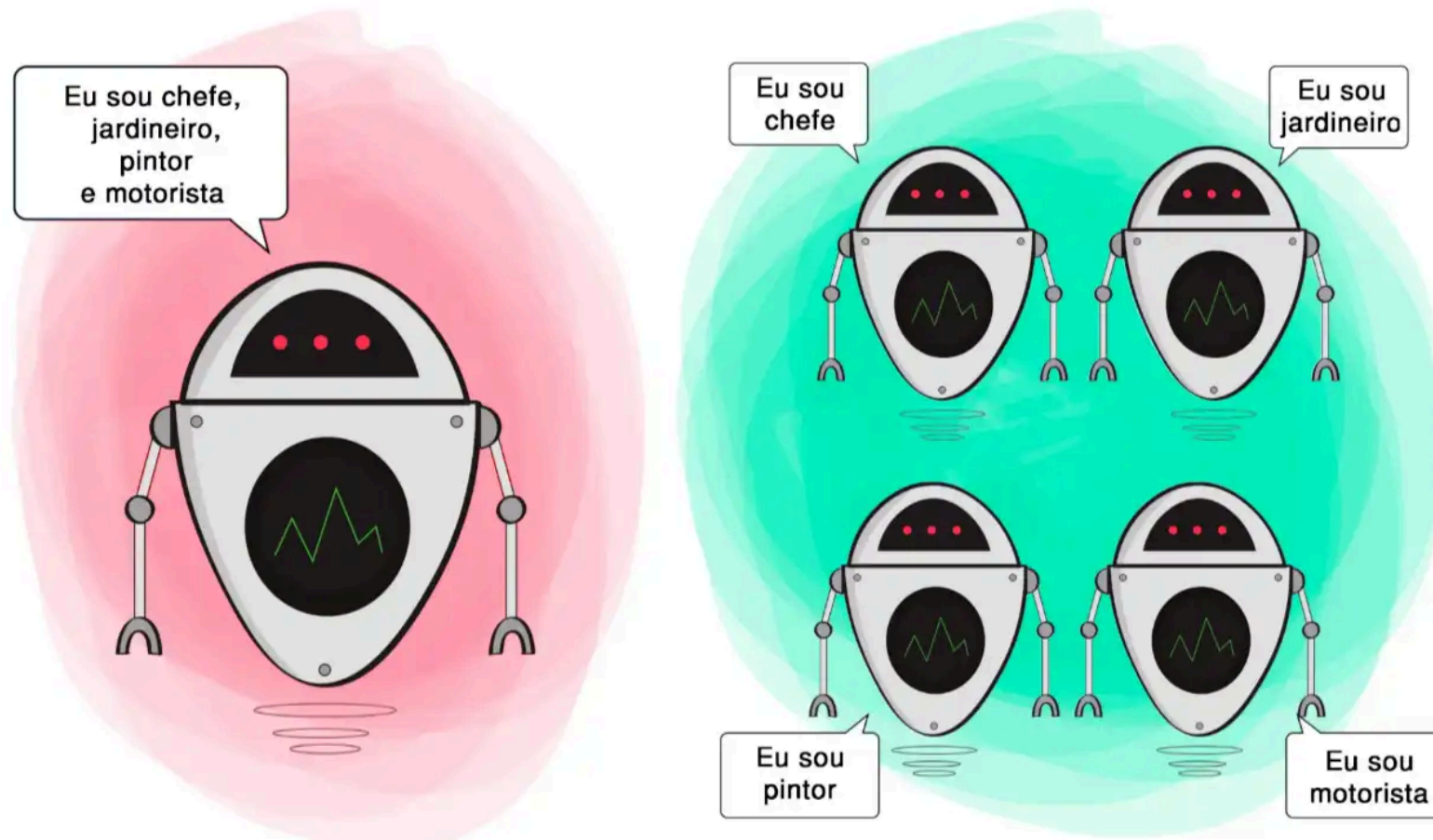
```
public class Pedido {  
    private Database db;  
    private EmailService email;  
  
    public double calcularTotal() { ... }  
    public void salvar() { db.save(this); }  
    public void enviarEmailConfirmacao() {  
        email.send("Pedido criado", "Seu pedido foi criado.");  
    }  
}
```

- Calcula, salva e envia email
- Múltiplos motivos de mudança

SRP – Bom exemplo

```
public class Pedido {  
    public double calcularTotal() { ... }  
}  
  
public class PedidoRepository {  
    public void salvar(Pedido pedido) { ... }  
}  
  
public class PedidoNotificador {  
    public void enviarConfirmacao(Pedido pedido) { ... }  
}
```

- Cada classe com um foco



Responsabilidade Única



O – Open/Closed



Entidades devem ser **abertas para extensão**,
mas **fechadas para modificação**.

- Adicionar novos comportamentos sem mexer em código já testado
- Usa muito **abstrações** e **polimorfismo**
- Não é “não modificar código”, mas sim “não modificar código desnecessariamente”.

OCP – Mau exemplo

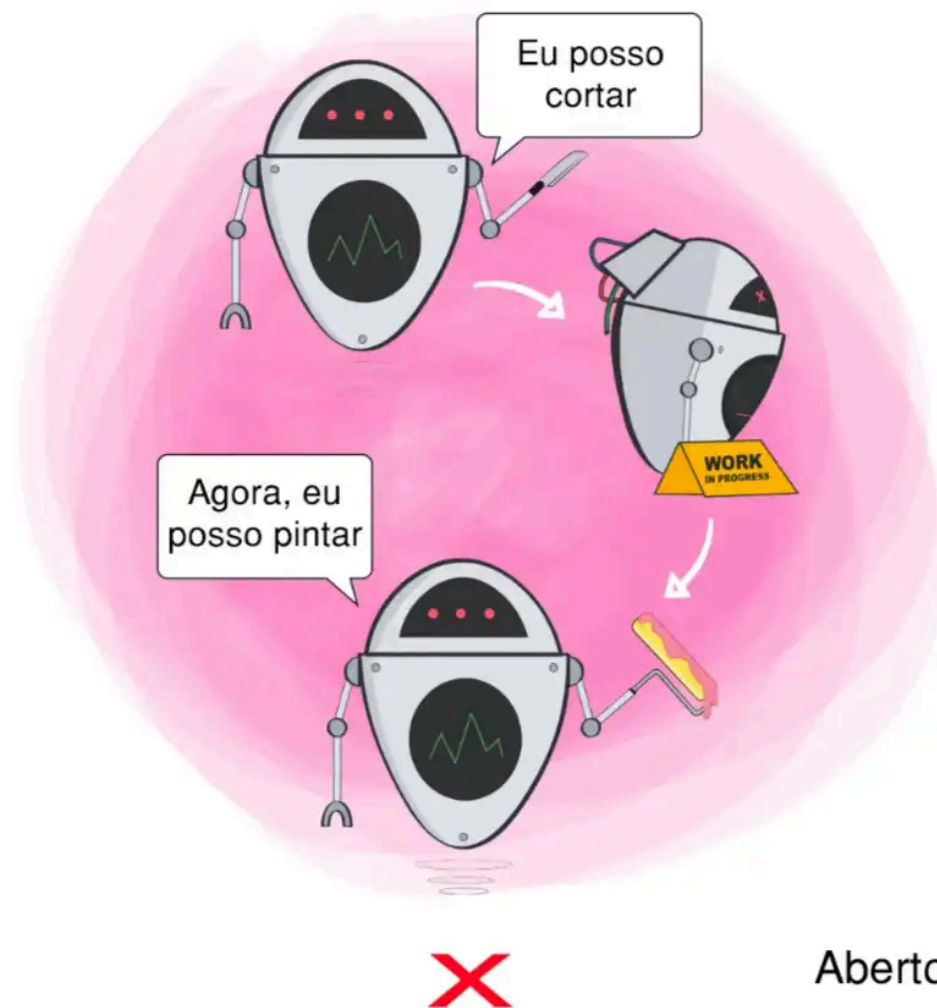
```
public class FolhaDePagamento {  
    public double calcularSalario(Funcionario f) {  
        if (f instanceof FuncionarioHorista) { ... }  
        else if (f instanceof FuncionarioMensalista) { ... }  
        // novos tipos exigem alterar este método  
    }  
}
```

- Cada novo tipo de funcionário mexe nessa classe

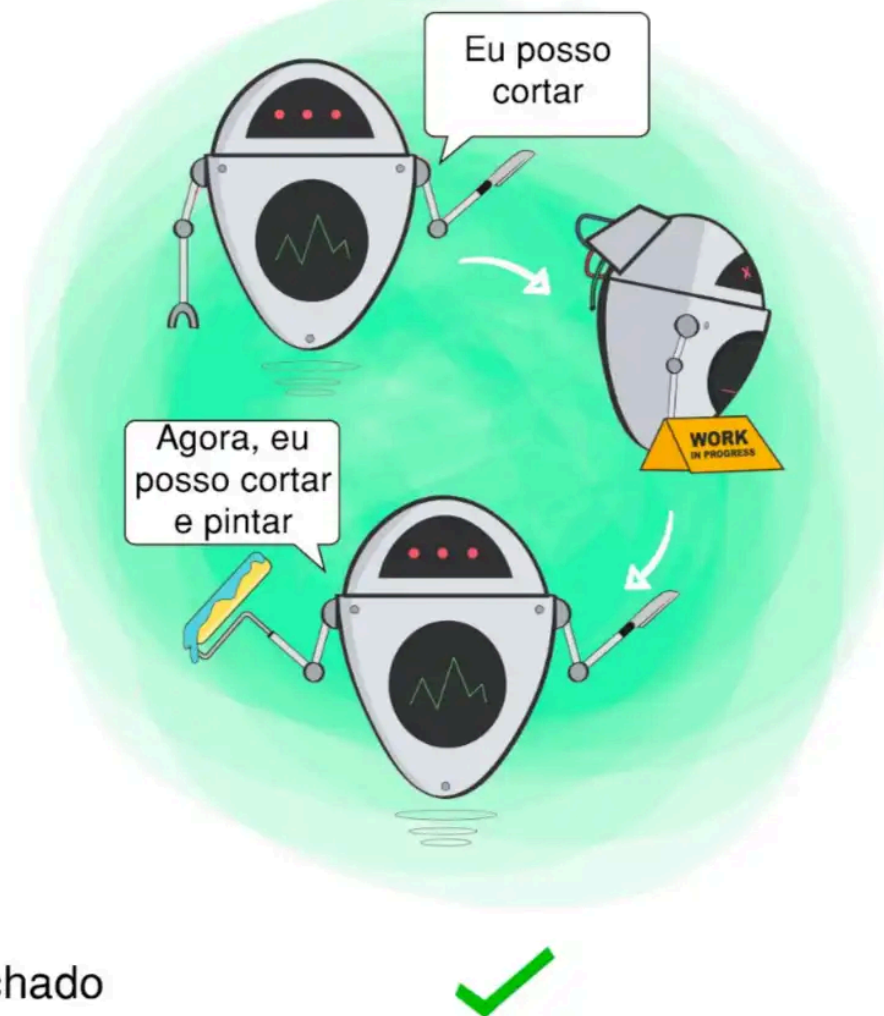
OCP – Bom exemplo

```
public abstract class Funcionario {  
    public abstract double calcularSalario();  
}  
  
public class FuncionarioHorista extends Funcionario {  
    public double calcularSalario() { ... }  
}  
  
public class FolhaDePagamento {  
    public double calcularSalario(Funcionario f) {  
        return f.calcularSalario();  
    }  
}
```

- Folha depende da **abstração** Funcionario



Aberto - Fechado

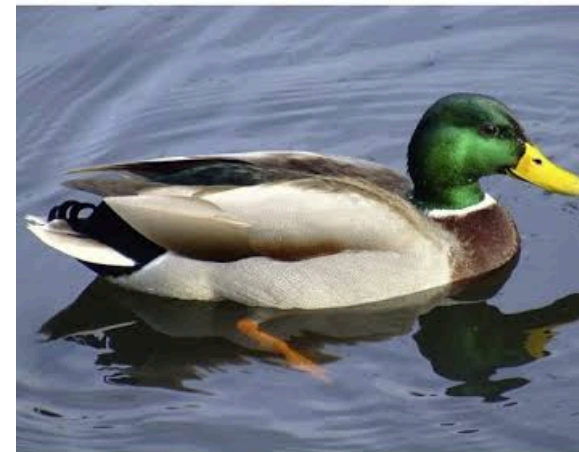


L – Liskov Substitution

Subclasses devem poder substituir a superclasse
sem quebrar o comportamento esperado.

Se **B** herda de **A**, objetos de **B** devem funcionar onde se espera **A**.

The Liskov Substitution Principle



If it looks like a duck and quacks like a duck but it needs batteries, you probably have the wrong abstraction.



LSP – Mau exemplo

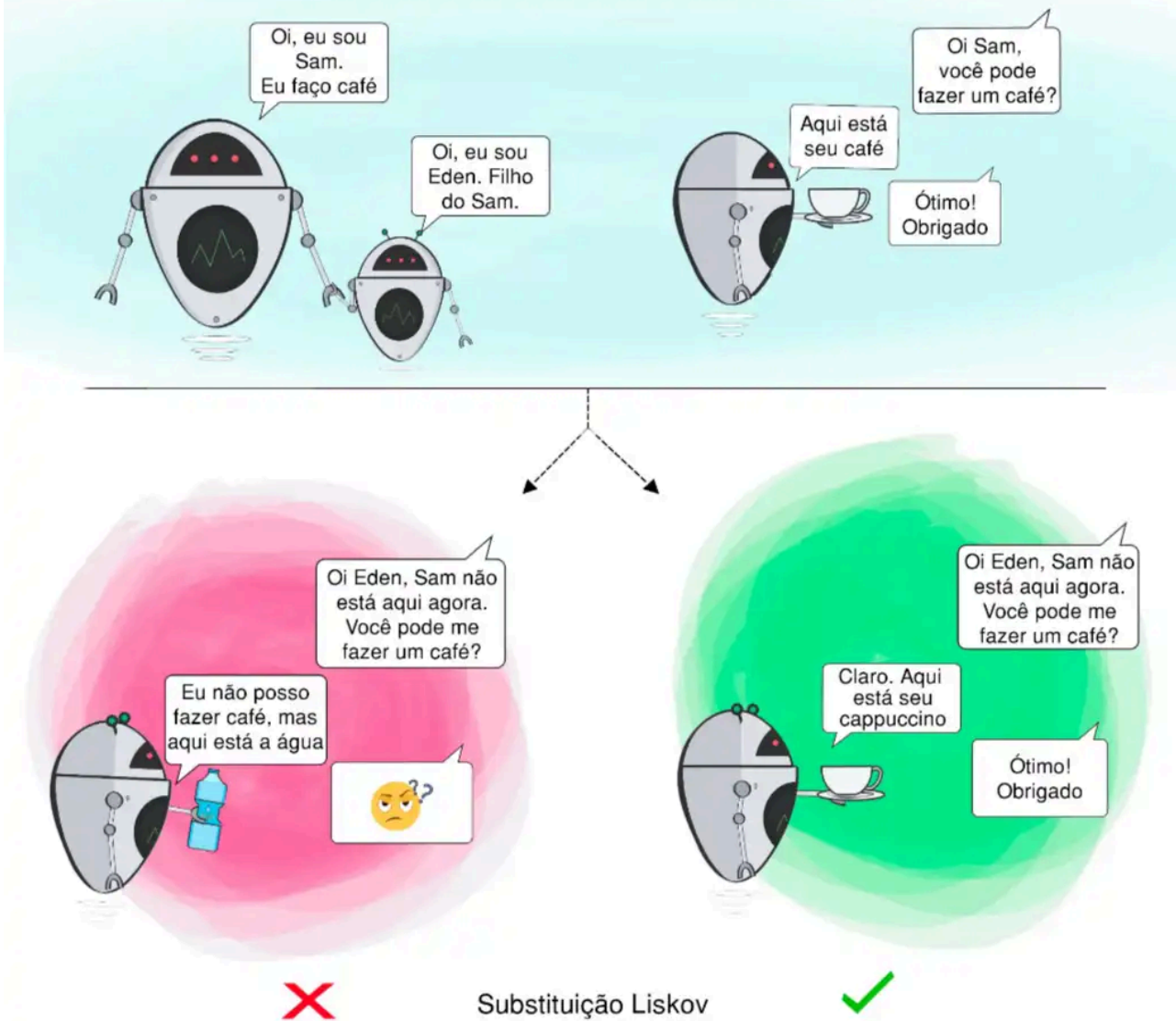
```
public class Retangulo {  
    public void setLargura(int l) { ... }  
    public void setAltura(int a) { ... }  
}  
  
public class Quadrado extends Retangulo {  
    @Override  
    public void setLargura(int l) {  
        super.setLargura(l);  
        super.setAltura(l);  
    }  
}
```

- Cliente espera largura e altura independentes
- Quadrado quebra essa expectativa

LSP – Bom exemplo

```
public abstract class Forma {  
    public abstract int calcularArea();  
}  
  
public class Retangulo extends Forma { ... }  
public class Quadrado extends Forma { ... }
```

- **Quadrado** não é forçado a se comportar como **Retangulo**





INTERFACE SEGREGATION PRINCIPLE

Don't force the client to depend on things they don't use.

I – Interface Segregation

Clientes não devem ser forçados a depender de métodos que não usam.

- Prefira interfaces **pequenas e específicas**

ISP – Mau exemplo

```
public interface Veiculo {  
    void acelerar();  
    void frear();  
    void voar();  
}  
  
public class Carro implements Veiculo {  
    public void voar() {  
        throw new UnsupportedOperationException();  
    }  
}
```

- Carro implementa algo que não faz sentido para ele

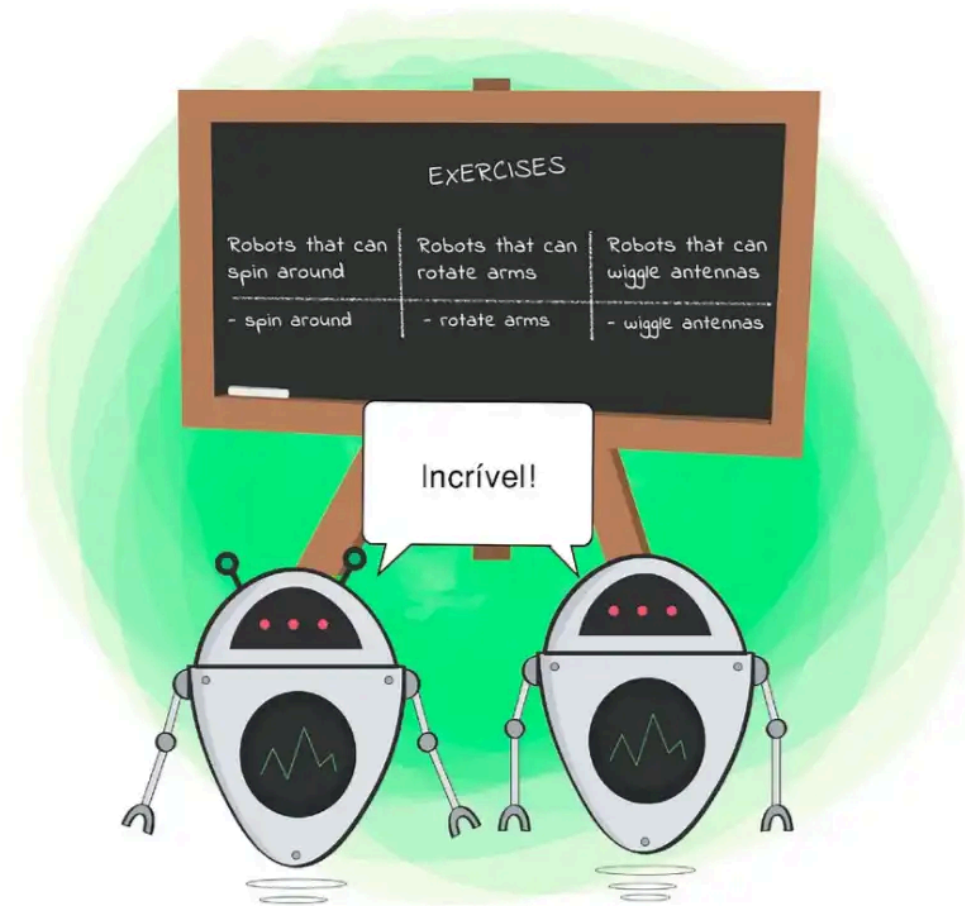
ISP – Bom exemplo

```
public interface VeiculoTerrestre {  
    void acelerar();  
    void frear();  
}  
  
public interface VeiculoAereo {  
    void voar();  
}  
  
public class Carro implements VeiculoTerrestre { ... }  
public class Aviao implements VeiculoAereo { ... }
```

- Cada cliente implementa só o que precisa



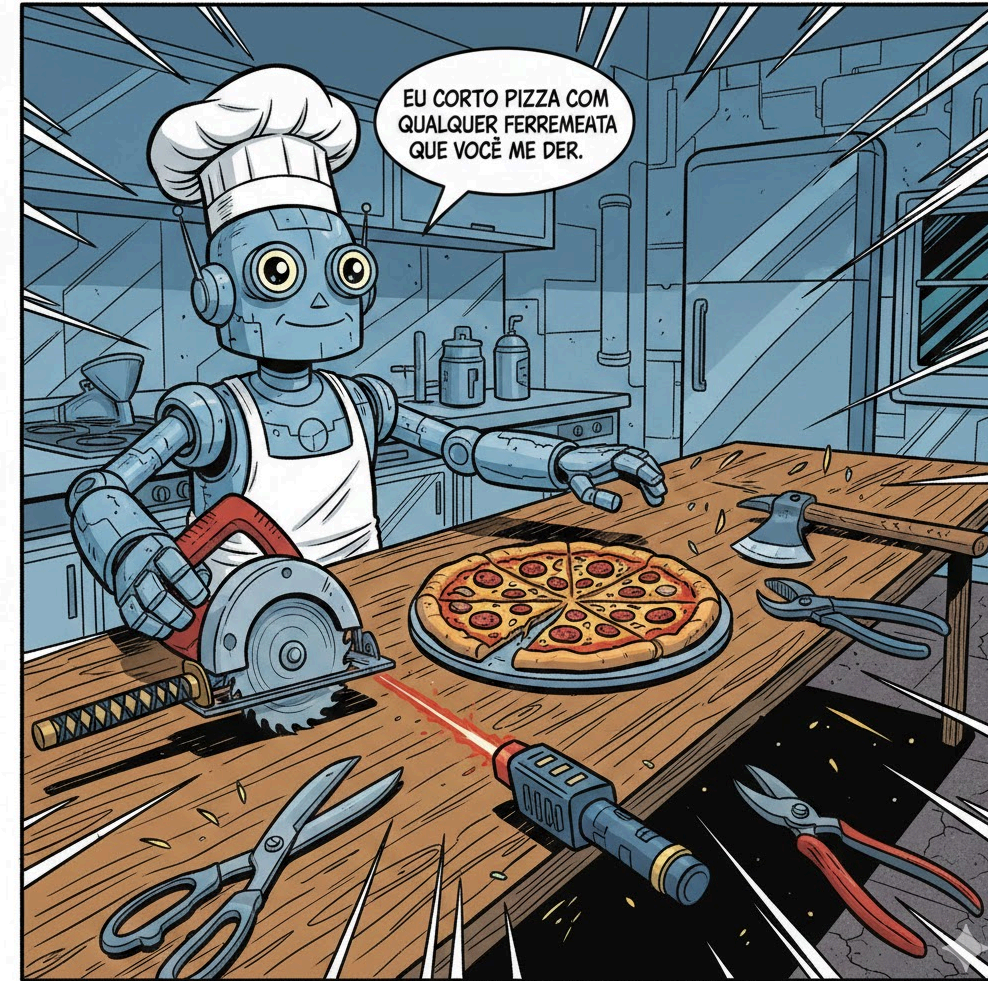
Segregação de Interface



D – Dependency Inversion

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de **abstrações**.

- Abstrações não devem depender de detalhes
- Detalhes devem depender de abstrações



DIP – Mau exemplo

```
public class Gmail {  
    public void enviarEmail(String dest, String msg) { ... }  
}  
  
public class ServicoEmail {  
    private Gmail gmail = new Gmail();  
  
    public void enviar(String dest, String msg) {  
        gmail.enviarEmail(dest, msg);  
    }  
}
```

- `ServicoEmail` depende diretamente de `Gmail`

DIP – Bom exemplo

```
public interface EmailService {  
    void enviarEmail(String dest, String msg);  
}  
  
public class Gmail implements EmailService {  
    public void enviarEmail(String dest, String msg) { ... }  
}
```

```
public class ServicoEmail {  
    private EmailService email;  
  
    public ServicoEmail(EmailService email) {  
        this.email = email;  
    }  
  
    public void enviar(String dest, String msg) {  
        email.enviarEmail(dest, msg);  
    }  
}
```

- Depende de **interface**, não da implementação concreta



✗



✓

Inversão de Dependência

SOLID e Design Patterns

- Muitos padrões de projeto ajudam a aplicar SOLID:
 - Strategy, Observer → OCP, DIP
 - Factory Method, Abstract Factory → DIP
 - Decorator → OCP
- SOLID é um **guia mental** útil

Outros Princípios

Além do SOLID, que é focado em **classes**, existem outros princípios de design de software:

- **DRY** – Don't Repeat Yourself
- **KISS** – Keep It Simple, Stupid
- **YAGNI** – You Aren't Gonna Need It



DRY – Don't Repeat Yourself

Cada pedaço de conhecimento deve ter uma **representação única** no sistema. Esse princípio foi formulado por Andy Hunt e Dave Thomas no livro *The Pragmatic Programmer*.

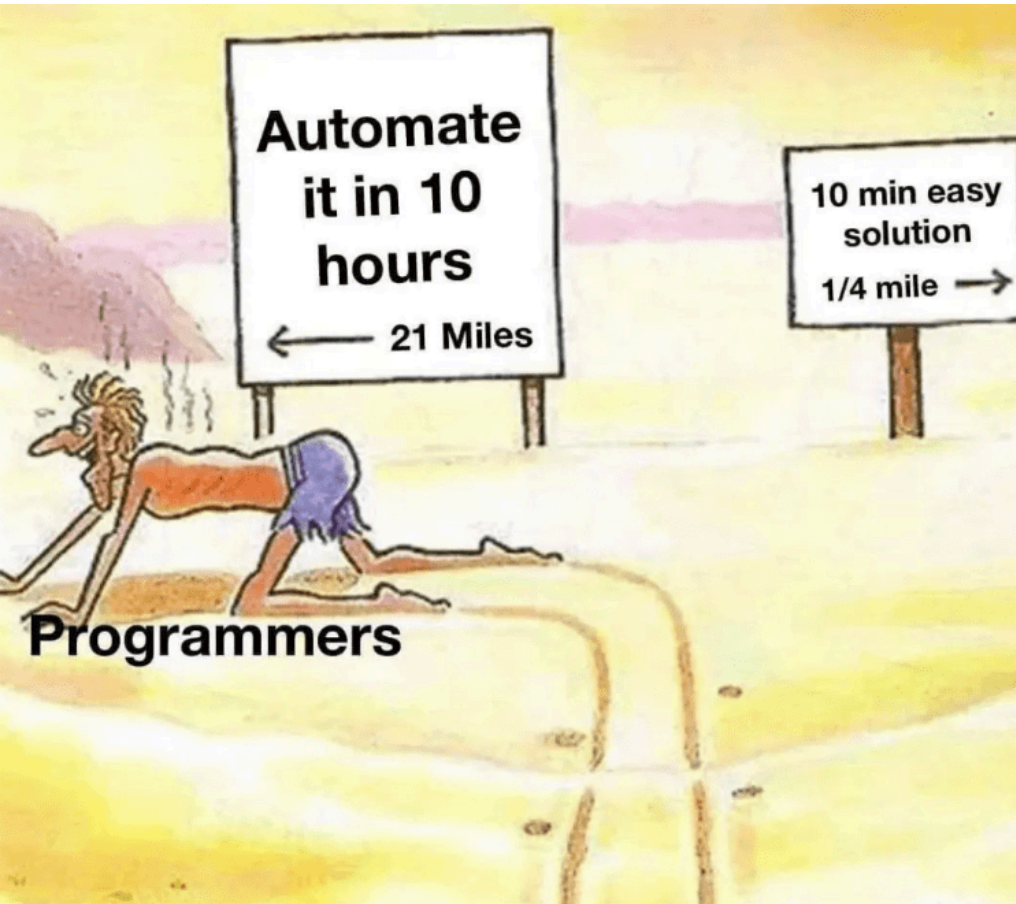
"Sempre que um sistema de software deve suportar um conjunto de alternativas, apenas um módulo no sistema deve conhecer a lista completa delas." - Bertrand Meyer

KISS – Keep It Simple, Stupid

Cunhado pela Marinha dos Estados Unidos, o princípio KISS enfatiza a importância de manter o design simples e direto. Sistemas complexos são mais difíceis de entender, testar e manter.

"A simplicidade é a sofisticação máxima" -
Leonardo da Vinci





YAGNI – You Aren't Gonna Need It

Popularizado pela metodologia Extreme Programming (XP), sugere que você não deve adicionar funcionalidades até que elas sejam realmente necessárias.

Sempre implemente as coisas quando você realmente precisar delas, nunca quando você apenas prevê que precisará delas. - Ron Jeffries

Resumo

- **SOLID**: 5 princípios para design de software orientado a objetos
 - *S (Single Responsibility)*: apenas uma razão para mudar
 - *O (Open/Closed)*: extensão antes de modificação
 - *L (Liskov Substitution)*: subclasses devem poder substituir superclasses
 - *I (Interface Segregation)*: interfaces pequenas e específicas
 - *D (Dependency Inversion)*: depender de abstrações
- **DRY**: Evitar duplicação de código
- **KISS**: Manter o design simples
- **YAGNI**: Não implementar funcionalidades desnecessárias