Programação Orientada a Objetos

OOP

Prof.: Armando Neto - kikobatery@hotmail.com - www.armandoneto.dev.br

Ementa

- Diagrama de Classe
- Objetos
- Classes
- Herança
- Polimorfismo
- Abstração
- Encapsulamento
- Conceitos de OOP (Acoplamento, Coesão, Associação, Agregação e Composição)
- Princípios SOLID

O que é?

- É o poder/mágica de trazer alguma coisa do mundo real para dentro da nossa linguagem. Também conhecido pelo termo de Abstração.

Classe

- É a representação de um Objeto do mundo real, por exemplo pessoa. Podemos definir uma pessoa no Delphi da seguinte maneira:

```
TPessoa = class
end;
```

Atributos e Propriedades

- Atributos e Propriedades são características únicas da classe. Seguindo o exemplo anterior uma pessoa tem Nome, Idade, Peso e etc...

```
TPessoa = class
   FNome: String;
   FIdade: Integer;
   FPeso: Double;
end;
```

Métodos

- Métodos são procedures ou funções coesas (que fazem sentido) para a classe:

```
TPessoa = class
   FNome: String;
   FSobreNome: String;
   FIdade: Integer;
   FPeso: Double;

   function RetornarNomeCompleto: String;
   procedure AlterarPeso(const aNovoPeso: Double);
end;
```

Encapsulamento

- Uma das vantagens da OOP é o encapsulamento, onde definimos o que deve estar visível ou não para outras classes do sistema:

*Existem também o published e protected.

```
TPessoa = class
  private
    FNome: String;
    FSobreNome: String;
    FNomeCompleto: String;
    FIdade: Integer;
    FPeso: Double;
  public
    function RetornarNomeCompleto: String;
    procedure AlterarPeso(const aNovoPeso: Double);
end;
```

Encapsulamento – “Getter and Setter” no Delphi

- O Delphi possui um get e set para encapsular seus atributos semelhante em outras linguagens:

```
TPessoa = class
  private
    FNome: String;
    FSobreNome: String;
    FNomeCompleto: String;
    FIdade: Integer;
    FPeso: Double;
  public
    function RetornarNomeCompleto: String;
    procedure AlterarPeso(const aNovoPeso: Double);

    property Nome: String read GetNome write SetNome(const aValue: String);
end;
```

Herança

- É a capacidade de uma classe “Herdar” característica de outra classe.
  Por exemplo, podemos criar uma classe **TPessoaFisica** herdando da classe **TPessoa**:

```
TPessoaFisica = class(TPessoa)
  private
    FCPF: String;
  public
     property CPF: String read GetCPF write SetCPF(const aValue: String);
end;
```

Sobrescrita - Override

- Recurso aplicado quando queremos sobrescrever algum método da classe herdada. Essa sobrescrita poderá ser total ou parcial:

```
TPessoaFisica = class(TPessoa)
  private
    FCPF: String;
  public
    function RetornarNomeCompleto: String; override;
    property CPF: String read GetCPF write SetCPF(const aValue: String);
end;


function TPessoaFisica.RetornarNomeCompleto: String;
begin
  inherited;

   FNomeCompleto := FNomeCompleto + " - " + FCPF;

   Result := FNomeCompleto;
end;
```

Sobrecarga - Overload

- Recurso aplicado quando temos dois métodos com o mesmo nome porém dependendo dos parâmetros realizam ações diferentes:

```
TPessoaFisica = class(TPessoa)
  private
    ...
  public
    function RetornarPeso: Double; overload;
    function RetornarPeso(const aTara: Double): Double; overload;
end;

function TPessoaFisica.RetornarPeso: Double;
begin
  Result := FPeso;
end;

function TPessoaFisica.RetornarPeso(const aTara: Double): Double;
begin
  Result := FPeso - aTara;
end;
```

Exercícios

- 01 – Introdução a Orientação a Objetos
- 02 – Herança

Construtores

- Os Construtores de Classes são utilizados para criar instâncias de Objetos. Uma Classe pode ter vários construtores, mas somente o Primário deverá retornar uma instância de um Objeto. Os demais construtores são apenas Secundários, pois irão fazer uso do construtor Primário. O construtor Primário, na maioria das vezes, sempre será aquele com mais parâmetros.

```
type
  TUser = class
  private
    FLogin: string;
    FPassword: string;
  public
    constructor Create(const Login, Password: string);
    constructor Create(const Login: string);
    function Login: string;
    function Password: string;
  end;

implementation

constructor TUser.Create(const Login, Password: string);
begin
  inherited Create;
  FLogin := Login;
  FPassword := Password;
end;

constructor TUser.Create(const Login: string);
begin
  Create(Login, '123456');
end;

function TUser.Login: string;
begin
  Result := FLogin;
end;

function TUser.Password: string;
begin
  Result := FPassword;
end;
```

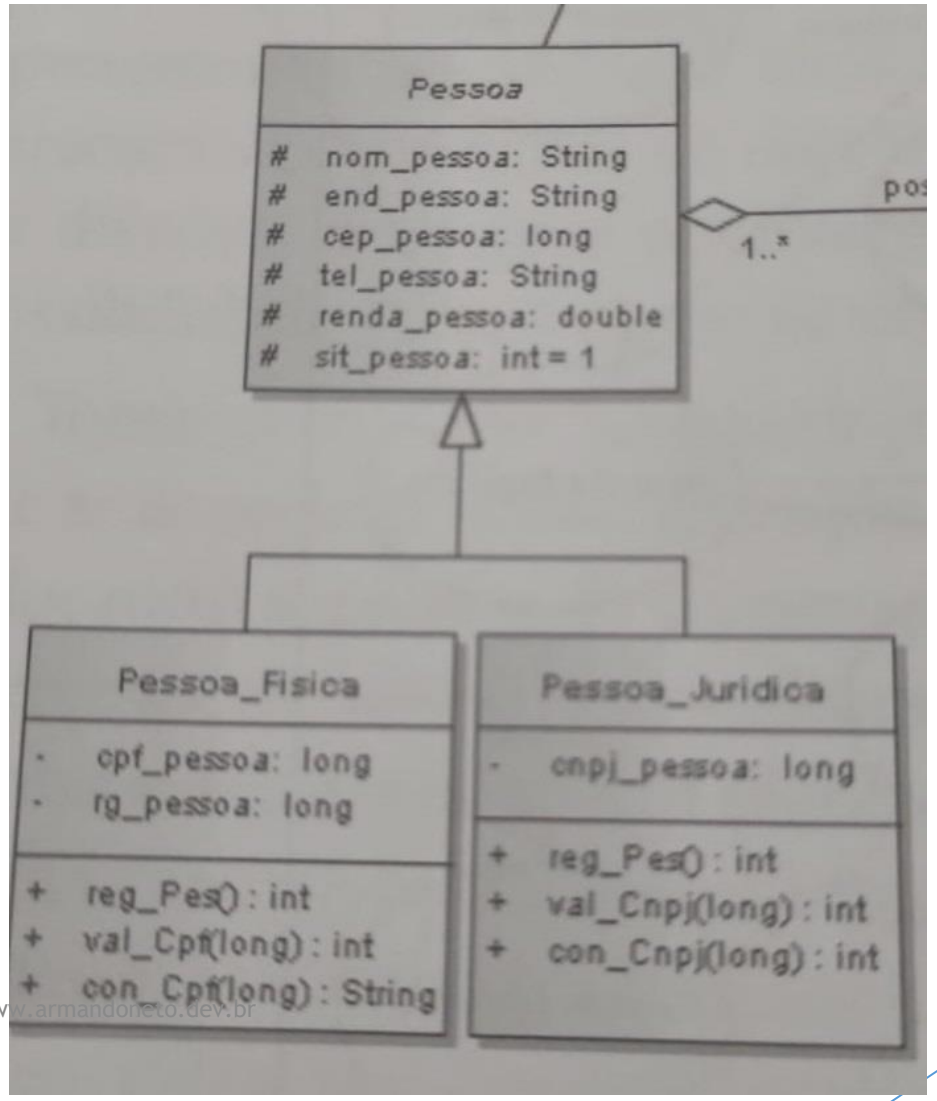
Exercícios

- 01 – Pessoa
- 02 – Conta Corrente
- 03 – Bomba Combustível

Diagramas de Classe

- O diagrama de classes é provavelmente o mais utilizado e é um dos mais importantes da UML (Linguagem de Modelagem Unificada). Serve de apoio para a maioria dos demais diagramas. Como o próprio nome diz, define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos que cada classe tem, além de estabelecer como as classes se relacionam e trocam informações entre si. A figura a seguir apresenta um exemplo desse diagrama

Diagramas de Classe



Diagramas de Classe

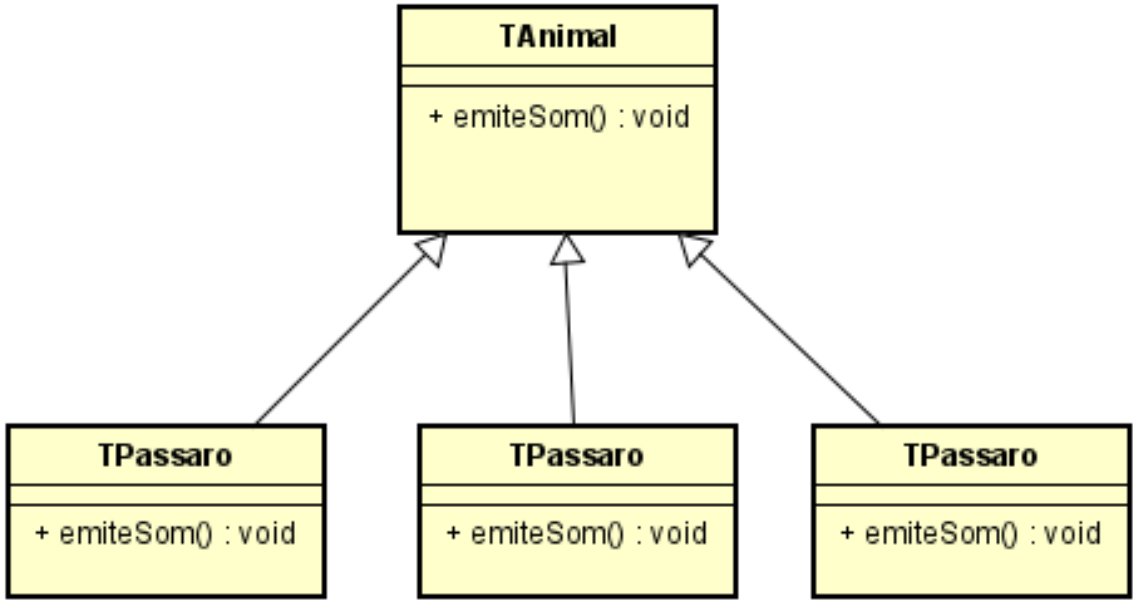
- 01 – Instalando o Astah
- 02 – Criando o primeiro diagrama de classe
- 03 – Criando a classe de acordo com o diagrama

Prof.: Armando Neto - kikobatery@hotmail.com - www.armandoneto.dev.br

Polimorfismo

- Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

Polimorfismo



Prática, Prática e Prática

- https://www.youtube.com/watch?v=mpP0fbO1fTA

Prática, Prática e Prática

- “Nós somos o que fazemos repetidamente. A excelência, portanto, não é um ato, mas um hábito”
- Pequenas e simples ações positivas, feitas consistentemente, criam grandes resultados

- Habilidades, não diplomas, definem hoje os melhores talentos.
- 9 em cada 10 profissionais são contratados pelo perfil técnico e demitidos pelo comportamental.

Prática, Prática e Prática

- “Segundo Malcolm Gladwell são necessárias nada menos do que 10 mil horas de prática para sermos especialistas em uma área” - https://www.psicologiamsn.com/2014/11/10-mil-horas-para-ser-um-especialista.html

- 8 horas por dia
- 5 vezes na semana (8 * 5 = 40 horas)
- 4 semanas por mês (40 * 4 = 160 horas)
- 12 meses ao ano (160 * 12 = 1.920 horas)
- Aproximadamente 5 anos (10.000 / 1.920 = 5,20 anos)

Abstração

- Um dos pilares da OOP (Abstração, Encapsulamento, Herança e Polimorfismo). A Abstração é utilizada para a definição de entidades do mundo real. Sendo onde são criadas as classes. Essas entidades são consideradas tudo que é real, tendo como consideração as suas características e ações.

Entidade	Características	Ações
Carro, Moto	Tamanho, cor, peso, altura	Acelerar, parar, ligar, desligar
Elevador	Tamanho, peso máximo	Subir, descer, escolher andar
Conta Banco	Saldo, limite, número	Depositar, sacar, ver extrato

Exercício

- Tema Livre – Faça a abstração de algo do mundo real.

Requisitos: Criar a classe, seus atributos de forma encapsulada e seus métodos.

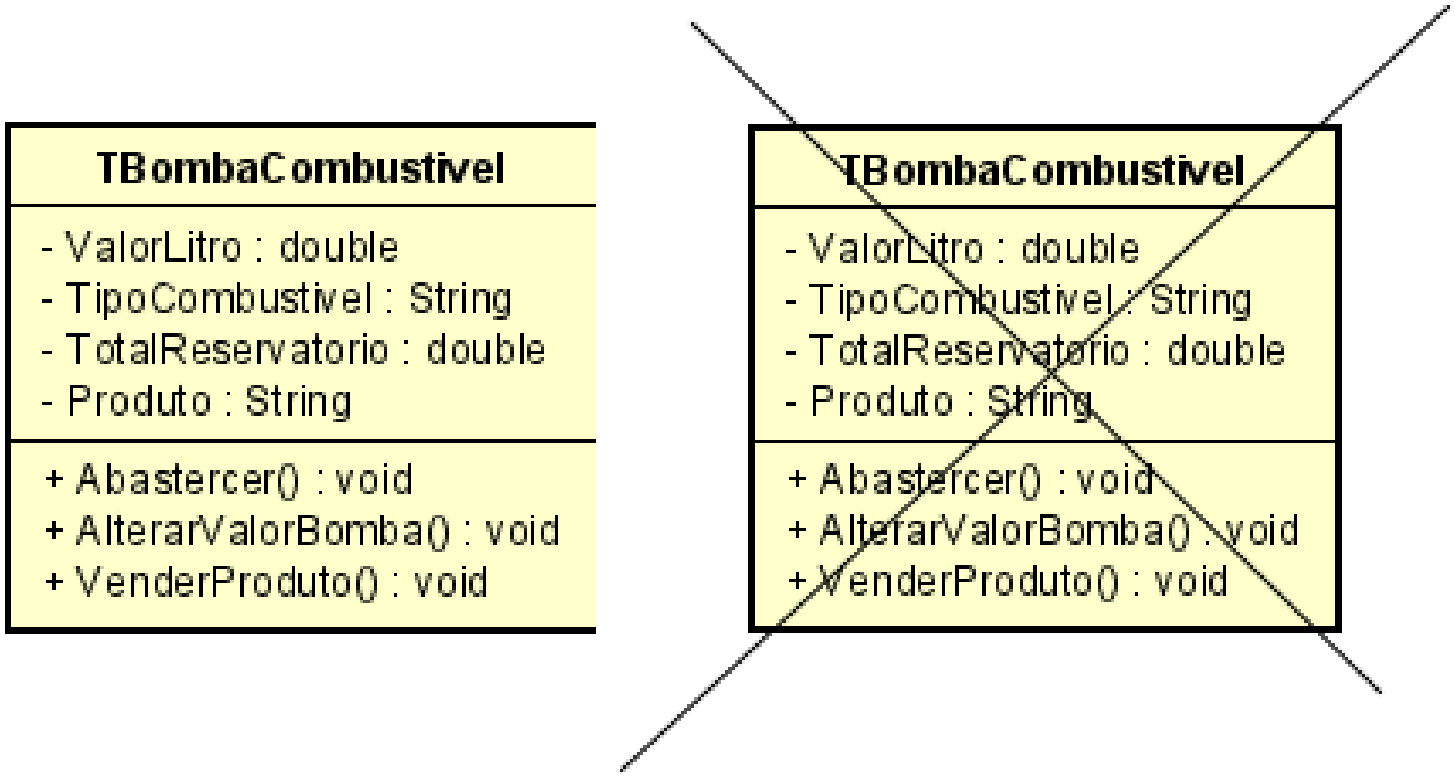
Conceitos de OOP

- Acoplamento
- Coesão
- Associação
- Agregação
- Composição

Coesão

- Coesão está ligado ao princípio da responsabilidade única que diz que uma classe deve ter apenas uma única responsabilidade e realizá-la de maneira correta, ou seja, uma classe não deve assumir responsabilidades que não são suas.

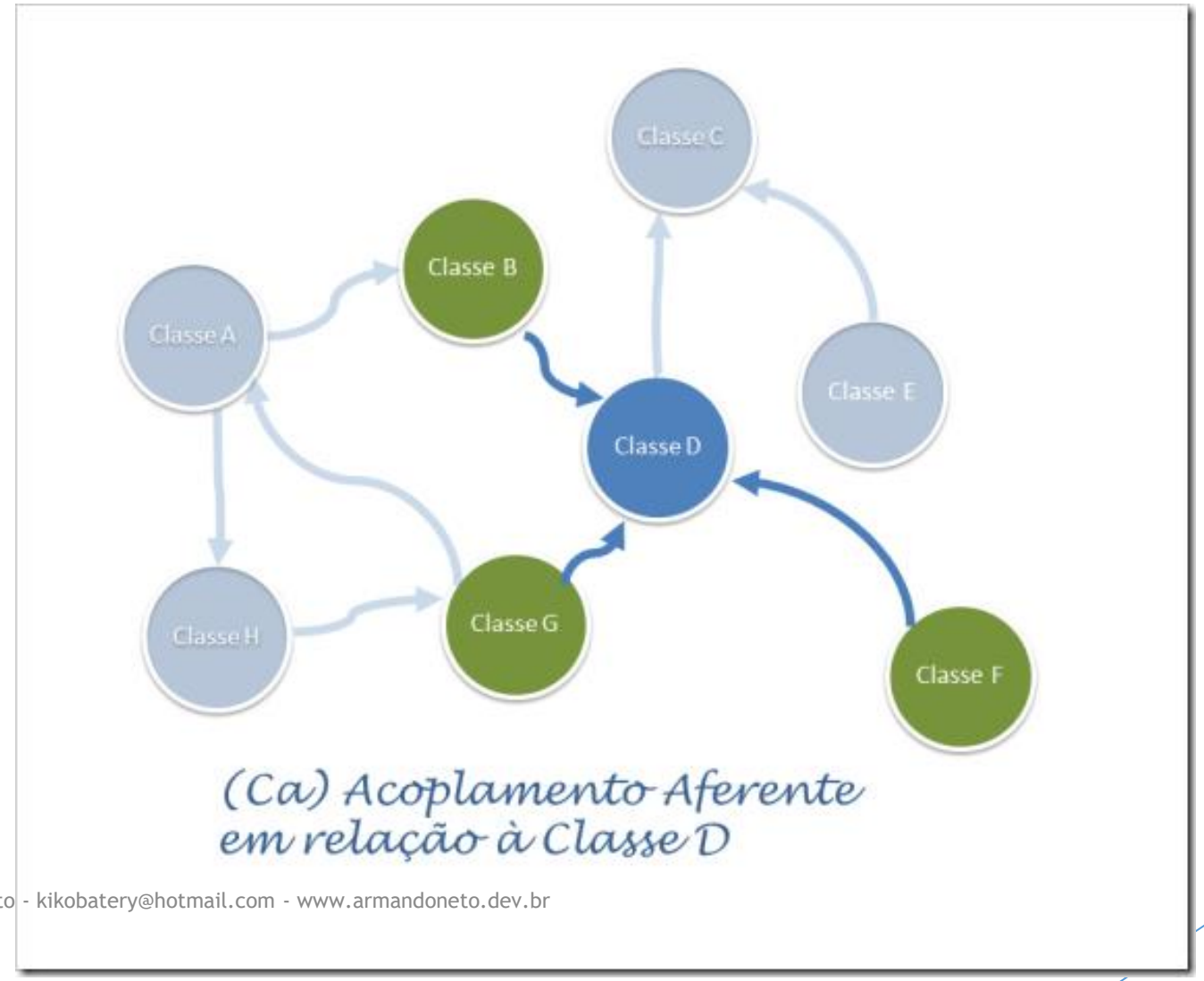
Coesão


TBombaCombustivel
- ValorLitro : double
- TipoCombustivel : String
- TotalReservatorio : double
- Produto : String
+ Abastercer() : void
+ AlterarValorBomba() : void
+ VenderProduto() : void
TBombaCombustivel
- ValorLitro : double
- TipoCombustivel : String
- TotalReservatorio : double
- Produto : String
+ Abastercer() : void
+ AlterarValorBomba() : void
+ VenderProduto() : void

Acoplamento

- Já o acoplamento é o quanto uma classe depende da outra para funcionar. E quanto maior for esta dependência entre ambas, dizemos que estas classes estão fortemente acopladas.

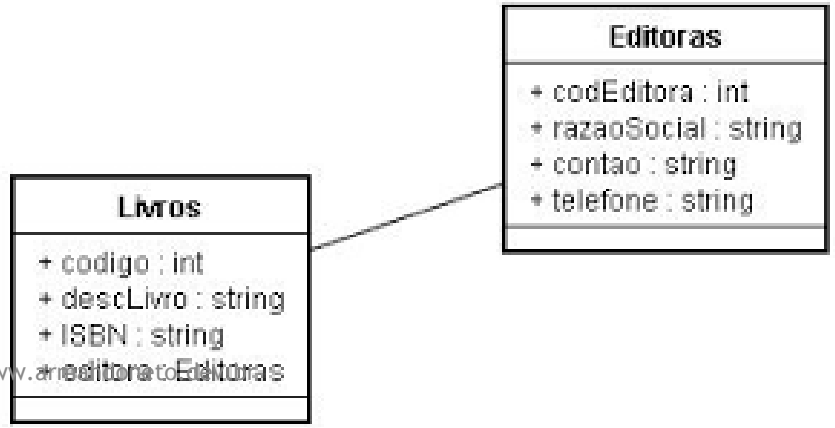
Acoplamento



Associação

- A associação entre dois objetos ocorre quando eles são completamente independentes entre si, mas eventualmente se relacionam. Pode ser considerada uma relação de Muitos para Muitos.

- Exemplo: A relação entre professor e alunos. Um aluno pode ter vários professores e um professor pode ter vários alunos. Um não depende do outro para existir.



Agregação

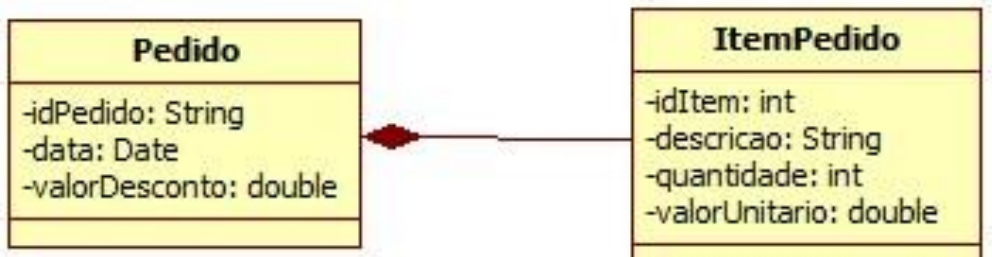
- A agregação não deixa de ser uma associação, mas existe uma exclusividade e determinados objetos só podem se relacionar a um objeto específico. É uma relação de Um para Muitos.

- Exemplo: Relação entre professores e os departamentos. Departamentos podem ter vários professores. E o professor só pode estar vinculado a um único departamento.

Composição

- A composição é uma agregação que possui dependência entre os objetos, ou seja, se o objeto principal for destruído, os objetos que o compõe não podem existir mais. Há a camada relação de morte.

- Exemplo: É a relação entre uma universidade e os departamentos. Além da universidade possuir vários departamentos, eles só podem existir se a universidade existir. Há uma dependência.



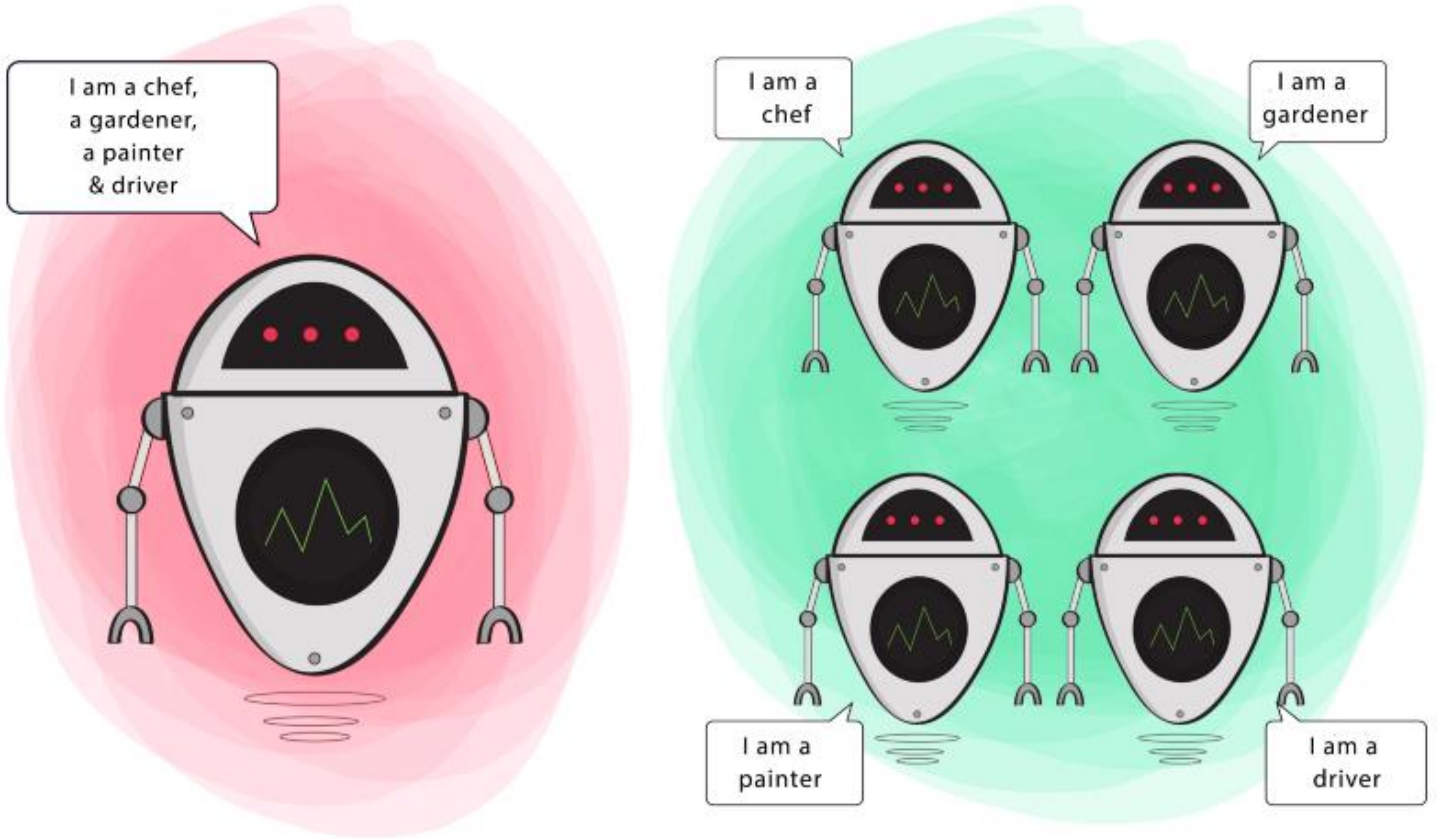
Princípios SOLID

- Single Responsibility Principle (Princípio da Responsabilidade Única)
- Open Closed Principle (Princípio Aberto Fechado)
- Liskov Substitution Principle (Princípio da Substituição de Liskov)
- Interface Segregation Principle (Princípio da Segregação de Interfaces)
- Dependency Inversion Principle (Princípio da Inversão de Dependência)

S – Responsabilidade Única

- Uma única responsabilidade – Coesa
- Com entidades independentes e isoladas, você consegue reaproveitar o código, refatorar, testes automatizados e gerar menos bugs
- Cada responsabilidade ficará isolado
- . Ex.: Do mau uso objeto do Cadastro acoplado ao objeto do Login, se fizer algum bug no Cadastro as pessoas deixam de logar no sistema.
- . Ex.: Quando lavamos roupa, basta uma meia colorida para manchar nossas roupas.

S – Responsabilidade Única



Single Responsibility

O – Aberto Fechado

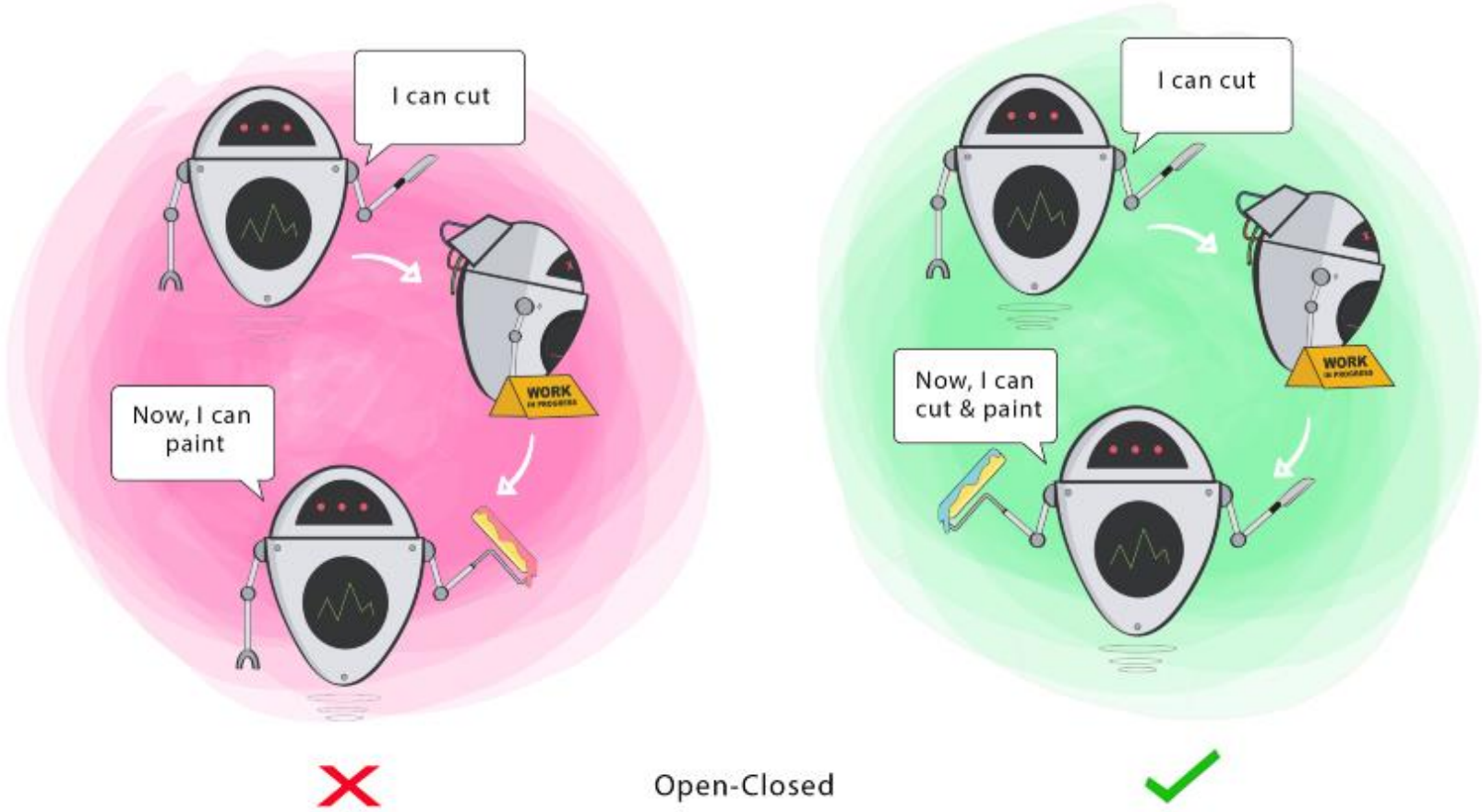
- Classes, Entidades e/ou Funções devem estarem abertas para extensões, mas fechadas para modificações.
- . Ex.: ProcessaPagamentos (cartao)

    . valida.Numero
    . Valida.vencimento
    . Valida.nome
    . Antifraude
    . cobrar

Se surgir o pagamento por boleto já “ferra” tudo. O correto é usar a abstração base. A abstração quem deve assumir o objeto que vai operar.

- O segredo é estender sua classe base.

O – Aberto Fechado



Open-Closed

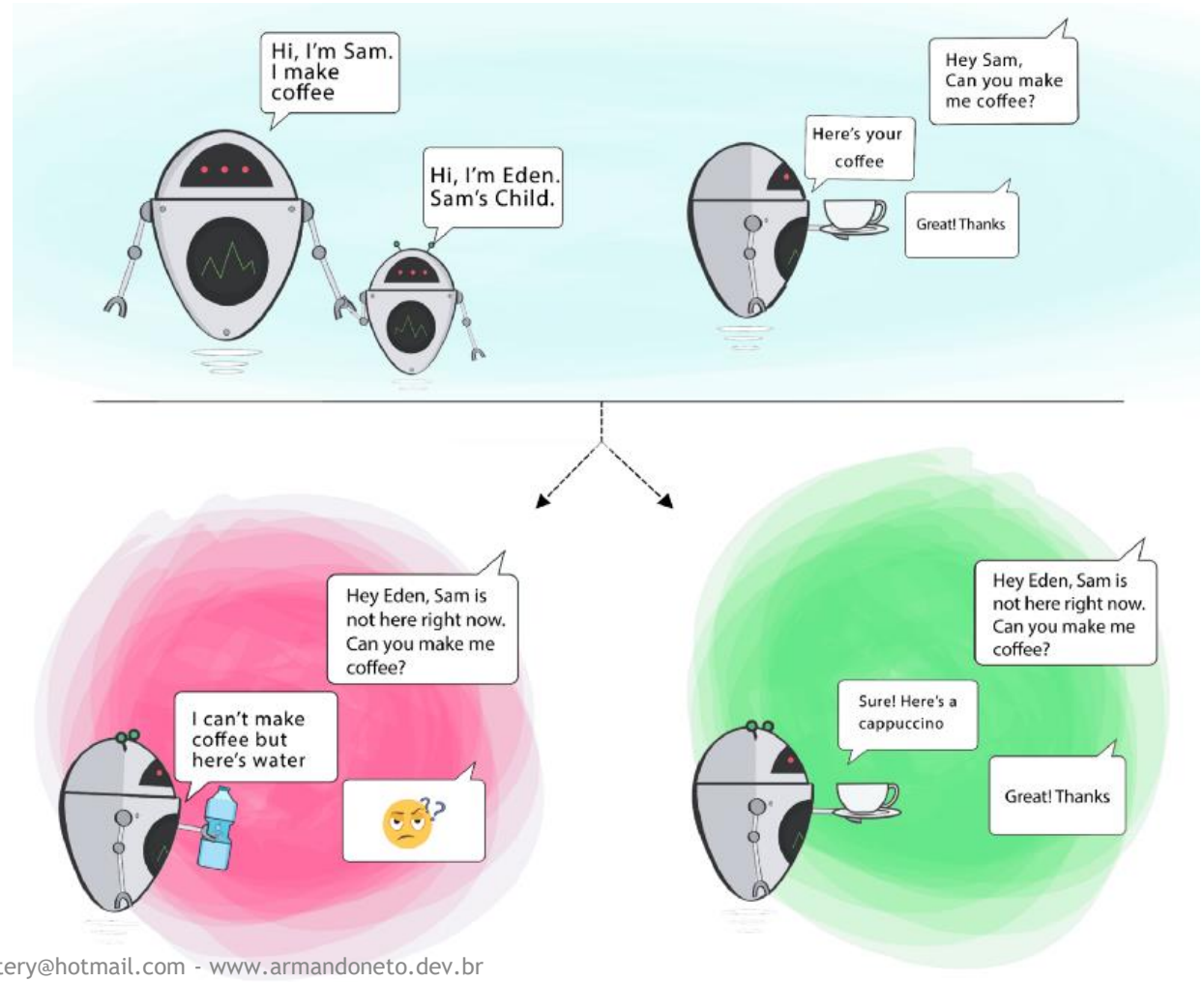
L – Substituição de Liskov

- Se temos uma classe e criarmos uma classe filha - herança. Essa Subclasse tem que ser capaz de realizar o que a classe pai realizada.
- Força ter suas classes na abstração mais correta e consistente.
- . Ex.: Ave()
  . bicar!
  . voar!

  PicaPau()
  . bicar!
  . voar!

  Pinguim()
  . bicar()
  #voar

L – Substituição de Liskov

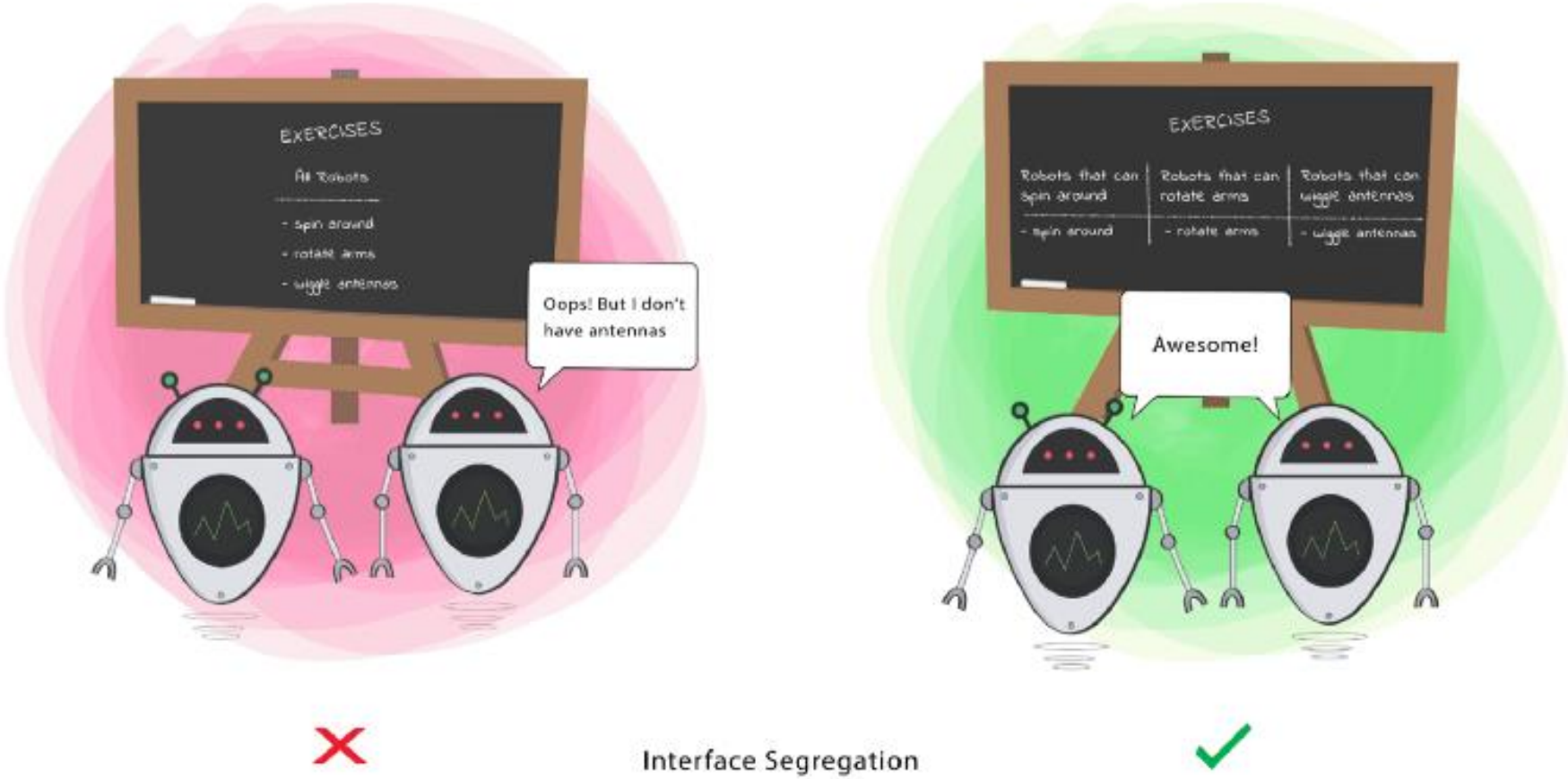


✗ Liskov Substitution ✓

I – Segregação de Interfaces

- Clientes não devem ser forçados a depender de métodos que eles não usam.
- Uma classe que é forçada a implementar uma interface com métodos que ela não vai precisar e que não faz sentido.

I – Segregação de Interfaces



D – Inversão de Dependência

- Um módulo não deve depender de implementação de outro módulo diretamente. Deve existir uma abstração no meio - uma interface.

D – Inversão de Dependência