

Programação Orientada a Objetos

Em Ruby, Java, C# e Python



Equipe:
Yasmin Maria Muniz de Oliveira
Victor Yan Pereira E Lima
Vanessa de Souza Câmara
Roberto Alves Neto
Vinícius Soares Da Costa
Vinicius Edwards Guimaraes Sousa



python™

Roteiro

1. História e propósito de cada linguagem

- 1.1. Ruby - Victor
- 1.2. Java - Vanessa
- 1.3. C# - Vinícius Soares
- 1.4. Python - Yasmin

3. Padrão de projeto Observer

- 3.1. Para que serve e como funciona - Vinícius Soares
- 3.2. Em Ruby - Victor
- 3.3. Em Java - Vanessa
- 3.4. Em C# - Vinícius Soares
- 3.5. Em Python - Yasmin

2. Orientação a objetos

- 2.1. História - Yasmin
- 2.2. Conceitos: classe e objeto; construtor; herança; associação, agregação e composição - Roberto
- 2.3. Exemplos em Ruby - Victor
- 2.4. Exemplos em Java - Roberto
- 2.5. Exemplos em C# - Roberto
- 2.6. Exemplos em Python - Roberto

História e propósito de cada linguagem



Ruby

Concebida em 24 de fevereiro de 1993 por Yukihiro Matsumoto, que pretendia criar uma nova linguagem que balanceava programação funcional com a programação imperativa.

"Eu queria uma linguagem de script que fosse mais poderosa do que Perl, e mais orientada a objetos do que Python. É por isso que eu decidi desenvolver minha própria linguagem".



Java

Java é uma linguagem que começou a ser criada em 1991, na Sun Microsystems. A ideia principal do Java era que aparelhos eletrônicos se comunicassem entre si. Essa linguagem foi a primeira a usar decodificadores de televisão interagindo em dispositivos portáteis e outros produtos eletrônicos e vem liderando o mercado desde a sua criação em termos de linguagem.

Algumas das principais características são:

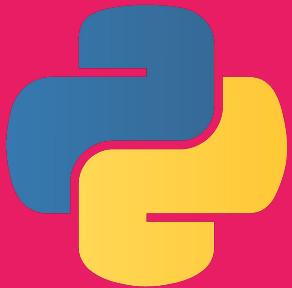
- Portabilidade
- Segurança
- Linguagem simples
- Alta performance
- Dinamismo
- Interpretada
- Distribuída
- Independente de plataformas
- Tipada



C#

C# foi desenvolvida por Anders Hejlsberg em 2000 como parte da iniciativa .NET da Microsoft e foi aprovada como padrão internacional em 2002. A linguagem foi concebida para a escrita de bibliotecas de classes do .NET framework, por esse motivo, C# foi desenvolvida como uma linguagem orientada a objetos "C-like".

— — —



Python

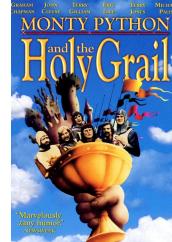
Foi criada em 1989 por Guido van Rossum na Holanda, para suceder a ABC e interagir com o sistema operacional Amoeba.

Nasceu para priorizar a legibilidade do código em vez da velocidade ou expressividade. Combina uma sintaxe concisa e clara com recursos poderosos padrão ou de terceiros.

De propósito geral, alto nível e multiparadigma: é orientada a objetos, imperativa, funcional e procedural.

A origem do nome vem do grupo de comédia britânico Monty Python.

— — —



Orientação a objetos

História

Surgiu da linguagem Simula 67 para fazer simulações de naves. A ideia era agrupar os tipos de naves em diversas classes de objetos, cada uma responsável por definir os seus próprios dados e comportamentos, para evitar colisões comportamentais entre elas.

Gradualmente, tornou-se um paradigma de programação em meados de 1990, devido à C++ e às interfaces gráficas.

Acrescentada a várias linguagens existentes, mas se consolidou em Java.

Conceitos

- Classe e objeto;
 - Construtor;
 - Herança;
 - Associação, agregação e composição.
-

Conceitos: Classes e Objetos

Objetos:

- Abstrações de conceitos do mundo real, seja algo físico ou imaginário;
- Possuem dados armazenados neles que o representam (**atributos**);
- Possuem atividades específicas que podem ser feitas através de solicitações (**métodos**);

Classe:

- Conjunto de objetos que possuem as mesmas características abstraiadas (atributos e métodos);
- Aqui é onde são implementadas as suas funcionalidades;

Um objeto é uma instância da classe, possui valores reais atribuídos aos campos abstratos:

```
um_endereco = Endereco("Manaus", "Parque Dez",  
                        "Av. Darcy Vargas",  
                        1200)
```

A classe define o comportamento dos atributos e métodos:

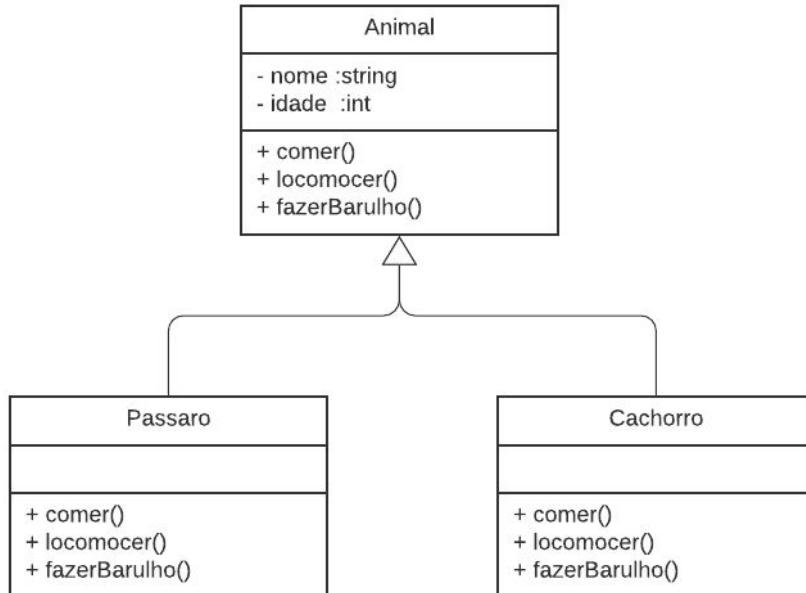
Endereço
- cidade :string - bairro :string - rua :string - casa :int
+ MostrarEndereco() : void

Conceitos: Herança

Herança:

- Quando classes diferentes dividem características em comum podemos usar herança;
- As classes filhas herdam todas as características da classe mãe, as modificam conforme sua necessidade e podem criar novas.

Tanto um pássaro quanto um cachorro comem e se movem, porém eles comem coisas diferentes e o pássaro se locomove voando enquanto o cachorro corre.



Conceitos: Associação, agregação e composição

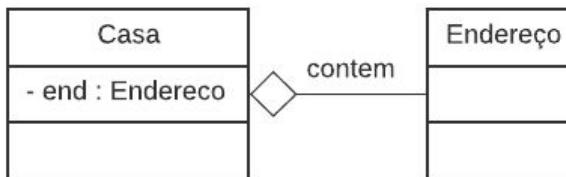
Associação:

- Classes diferentes e independentes entre si se conhecem e se comunicam;



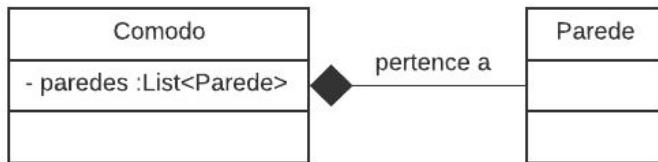
Agregação:

- Tipo especial de associação;
- As informações de um objeto precisam ser complementadas com outro;
- A existência da parte faz sentido sem o todo.



Composição:

- Tipo especial de associação;
- As informações de um objeto precisam ser complementadas com outro;
- A existência da parte não faz sentido sem o todo.





Ruby

Classe, Objeto e Construtor

```
1 #Herança
2 class Pessoa
3   def nome
4     @nome
5   end
6
7   def nome=( value )
8     @nome = value
9   end
10
11 > def idade...
13   end
14
15 > def idade=( value )...
17   end
18
19   # constructor
20   def initialize(nome, idade)
21     @nome = nome
22     @idade = idade
23   end
24 end
```



Ruby Herança

```
26  class PessoaFisica < Pessoa
27    def cpf
28      @cpf
29    end
30
31    def cpf=( value )
32      @cpf = value
33    end
34
35    # constructor
36  >  def initialize(nome, idade, cpf)...
37    end
38
39
40
41    def to_s()
42      "Nome: %s -- Idade: %d -- CPF: %d"
43      % [ @nome, @idade, @cpf ]
44    end
45  end
```



Ruby - Associação

```
74 < class Professor  
75  
76 >   def nome...  
77   end  
78  
79 >   def nome=( value )...  
80   end  
81  
82 >   def alunos...  
83   end  
84  
85 >   def alunos=( value )...  
86   end  
87  
88 >   def addAluno(aluno)  
89 |     @alunos << aluno  
90 |     aluno.addProfessor(self)  
91 end  
92  
93 # constructor  
94 < class Aluno  
95 >   def matricula...  
96   end  
97  
98 >   def initialize(nome)  
99 |     @nome = nome  
100|     @alunos = []  
101end  
102end
```

```
104 < class Aluno  
105 >   def matricula...  
106   end  
107  
108 >   def matricula=( value )...  
109   end  
110  
111 >   # Atribuios Nome, Professor, ...  
112  
113 >   def addProfessor(professor)  
114 |     @professores << professor  
115 end  
116  
117 >   def to_s()  
118 |     "%d Nome: %s" % [@matricula, @nome]  
119 end  
120  
121 >   # constructor  
122 >   def initialize(matricula, nome)  
123 |     @matricula = matricula  
124 |     @nome = nome  
125 |     @professores = []  
126 end  
127  
128 >   end  
129 end
```



- Não existe uma dependência de existência nem por parte de **Professor** ou **Aluno**.
- Um **professor** pode ter vários **alunos**.
- Um **aluno** pode ter mais de um **professor**.

```
professor1 = Professor.new('Jucimar')
professor2 = Professor.new('Fabio')

aluno1 = Aluno.new(1001, 'Roberto')
aluno2 = Aluno.new(1002, 'Vinicius')
aluno3 = Aluno.new(1003, 'Vanessa')
aluno4 = Aluno.new(1004, 'Victor')
aluno5 = Aluno.new(1005, 'Yasmin')

professor1.addAluno(aluno1)
professor1.addAluno(aluno2)
professor1.addAluno(aluno3)
professor1.addAluno(aluno4)

professor2.addAluno(aluno2)
professor2.addAluno(aluno3)
professor2.addAluno(aluno4)
professor2.addAluno(aluno5)

puts 'Professor(a): %s' % [professor1.nome]
puts (professor1.alunos)
puts
puts 'Professor(a): %s' % [professor2.nome]
puts (professor2.alunos)
puts
```



Ruby

Agregação

- Não existe dependência de existência.
- Porém existe uma relação ‘A possui B’.
- No caso, o automóvel continuará existindo sem o motorista.

```
173 < class Motorista
174
175 <   def nome
176   |   @nome
177   end
178
179 <   def nome=( value )
180   |   @nome = value
181   end
182
183 <   def automovel
184   |   @automovel
185   end
186
187 <   def automovel=( value )
188   |   @automovel = value
189   end
190
191   # constructor
192 <   def initialize(nome, automovel)
193   |   @nome = nome
194   |   @automovel = automovel
195   end
196 end
```



Ruby - Composição

```
132 class Motor
133   def potencia
134     @potencia
135   end
136
137   def potencia=( value )
138     @potencia = value
139   end
140
141 # constructor
142 def initialize(potencia)
143   @potencia = potencia
144 end
145 end
```

```
147 class Automovel
148
149 def modelo
150   @modelo
151 end
152
153 def modelo=( value )
154   @modelo = value
155 end
156
157 def motor
158   @motor
159 end
160
161 def motor=( value )
162   @motor = value
163 end
164
165 # constructor
166 def initialize(modelo, potenciaMotor)
167   @modelo = modelo
168   @motor = Motor.new(potenciaMotor)
169 end
170 end
```

- Um depende do outro para existir. Não existe **Automóvel** sem **Motor** e vice-versa. O **Motor** deixará de existir junto ao **Automóvel**



Java

Classe, Objeto e Construtor

```
1 // Classe
2 class Endereco {
3     public String cidade;
4     public String bairro;
5     public String rua;
6     public int casa;
7
8     // Construtor
9     public Endereco(String cidade, String bairro, String rua, int casa) {
10         this.cidade = cidade;
11         this.bairro = bairro;
12         this.rua = rua;
13         this.casa = casa;
14     }
15 }
```

```
1 // Objeto
2 Endereco endereco1 = new Endereco("Manaus", "Parque Dez", "Av. Darcy Vargas", 1200);
```

— — —



Java

Herança

```
1 // Herança
2 abstract class ACasa {
3     public List<Quarto> quartos;
4     public List<Dono> donos;
5     public Endereco endereco;
6
7     public abstract void exibirEndereco();
8
9     public abstract void addDono(Dono dono);
10
11    public abstract void exibirDonos();
12 }
```

```
1 class Casa extends ACasa {
2
3     public Casa() {
4         this.quartos = new ArrayList<Quarto>();
5         this.donos = new ArrayList<Dono>();
6     }
7     public Casa(Endereco endereco) {
8         this.quartos = new ArrayList<Quarto>();
9         this.donos = new ArrayList<Dono>();
10        this.endereco = endereco;
11    }
12
13    @Override
14    public void exibirEndereco() {
15        String textoEnd = "\nENDERECO\nCidade: " + this.endereco.cidade + "\nBairro: ";
16        textoEnd += this.endereco.bairro + "\nRua: " + this.endereco.rua;
17        textoEnd += "\n\n" casa: " + this.endereco.casa;
18        System.out.println(textoEnd);
19    }
20
21    @Override
22    public void addDono(Dono dono) {
23        this.donos.add(dono);
24    }
25
26    @Override
27    public void exibirDonos() {
28        System.out.println("\nOs donos são:");
29        for (Dono dono : this.donos) {
30            System.out.println(dono.nome);
31        }
32    }
33 }
```



Java - Associação

```
1 // Associação
2 class Dono {
3     public String nome;
4     public List<Casa> casas;
5
6     public Dono(String nome) {
7         this.casas = new ArrayList<Casa>();
8         this.nome = nome;
9     }
10    public void addCasa(Casa casa) {
11        this.casas.add(casa);
12    }
13 }
14 }
```

```
1 public class App {
2     public static void main(String[] args) throws Exception {
3         Dono dono1 = new Dono("Joao");
4         Dono dono2 = new Dono("Lari");
5         Dono dono3 = new Dono("Mario");
6
7         Endereco enderecol = new Endereco("Manaus", "Parque Dez",
8             "Av. Darcy Vargas", 1200);
9
10        CasaBuilder planta = new PlantaCasa(enderecol);
11        Casa casa = planta.getcasa();
12
13        casa.addDono(dono1);
14        casa.addDono(dono2);
15        casa.addDono(dono3);
16
17        casa.exibirDonos();
18    }
19 }
20 }
```

```
1 // Classe
2 class Casa extends ACasa {
3
4     public Casa() {
5         this.quartos = new ArrayList<Quarto>();
6         this.donos = new ArrayList<Dono>();
7     }
8     public Casa(Endereco endereco) {
9         this.quartos = new ArrayList<Quarto>();
10        this.donos = new ArrayList<Dono>();
11        this.endereco = endereco;
12    }
13
14    @Override
15    public void exibirEndereco() {
16        String textoEnd = "\nENDERECO\nCidade: " + this.endereco.cidade + "\nBairro: ";
17        textoEnd += this.endereco.bairro + "\nRua: " + this.endereco.rua;
18        textoEnd += "\nNº casa: " + this.endereco.casa;
19        System.out.println(textoEnd);
20    }
21
22    @Override
23    public void addDono(Dono dono) {
24        this.donos.add(dono);
25    }
26
27    @Override
28    public void exibirDonos() {
29        System.out.println("\nOs donos são:");
30        for (Dono dono : this.donos) {
31            System.out.println(dono.nome);
32        }
33    }
34 }
```



Java

Agregação

```
1 // Agregação
2 class Endereco {
3     public String cidade;
4     public String bairro;
5     public String rua;
6     public int casa;
7
8     public Endereco(String cidade, String bairro, String rua, int casa) {
9         this.cidade = cidade;
10        this.bairro = bairro;
11        this.rua = rua;
12        this.casa = casa;
13    }
14 }
```

```
1 class Casa extends ACasa {
2
3     public Casa() {
4         this.quartos = new ArrayList<Quarto>();
5         this.donos = new ArrayList<Dono>();
6     }
7     public Casa(Endereco endereco) {
8         this.quartos = new ArrayList<Quarto>();
9         this.donos = new ArrayList<Dono>();
10        this.endereco = endereco;
11    }
12
13    @Override
14    public void exibirEndereco() {
15        String textoEnd = "\nENDERECO\nCidade: " + this.endereco.cidade + "\nBairro: ";
16        textoEnd += this.endereco.bairro + "\nRua: " + this.endereco.rua;
17        textoEnd += "\nNº casa: " + this.endereco.casa;
18        System.out.println(textoEnd);
19    }
20
21    @Override
22    public void addDono(Dono dono) {
23        this.donos.add(dono);
24    }
25
26    @Override
27    public void exibirDonos() {
28        System.out.println("\nOs donos são:");
29        for (Dono dono : this.donos) {
30            System.out.println(dono.nome);
31        }
32    }
33 }
```



Java Composição

```
1 // Composição
2 class Parede {
3     public String cor;
4
5     public Parede(String cor) {
6         this.cor = cor;
7     }
8 }
9
10 class Porta {
11     public int idQuartoAtual;
12     public int idQuartoDestino;
13
14     public Porta(int idQuartoAtual, int idQuartoDestino) {
15         this.idQuartoAtual = idQuartoAtual;
16         this.idQuartoDestino = idQuartoDestino;
17     }
18 }
```

```
1 class Quarto {
2     public int id;
3     public Parede paredeEsq;
4     public Parede paredeDir;
5     public Parede paredeFrente;
6     public Parede paredeAtras;
7     public Porta porta;
8
9     public Quarto(int id, Parede paredeEsq, Parede paredeDir,
10                  Parede paredeFrente, Parede paredeAtras) {
11         this.id = id;
12         this.paredeEsq = paredeEsq;
13         this.paredeDir = paredeDir;
14         this.paredeFrente = paredeFrente;
15         this.paredeAtras = paredeAtras;
16     }
17 }
```



Classe, Objeto e Construtor

C#

- □ ×

```
1 // Classe
2 public class Endereco
3 {
4     public string Cidade { get; set; }
5     public string Bairro { get; set; }
6     public string Rua { get; set; }
7     public int Casa { get; set; }
8
9     // Construtor
10    public Endereco(string cidade, string bairro, string rua, int casa)
11    {
12        this.Cidade = cidade;
13        this.Bairro = bairro;
14        this.Rua = rua;
15        this.Casa = casa;
16    }
17 }
```

```
1 // Objeto
2 Endereco endereco1 = new Endereco("Manaus", "Parque Dez", "Av. Darcy Vargas", 1200);
```

— — —



C#

Herança

```
1 // Herança
2 abstract class ACasa
3 {
4     public abstract List<Quarto> Quartos { get; set; }
5     public abstract List<Dono> Donos { get; set; }
6     public abstract Endereco Endereco { get; set; }
7
8     public abstract void exibirEndereco();
9     public abstract void exibirDonos();
10    public abstract void addDono(Dono dono);
11 }
```

```
1 // Classe
2 class Casa : ACasa
3 {
4
5     public override List<Quarto> Quartos { get; set; }
6     public override List<Dono> Donos { get; set; }
7     public override Endereco Endereco { get; set; }
8
9     public Casa() {
10         Quartos = new List<Quarto>();
11         Donos = new List<Dono>();
12     }
13     public Casa(Endereco endereco) {
14         Quartos = new List<Quarto>();
15         Donos = new List<Dono>();
16         this.Endereco = endereco;
17     }
18
19     public override void exibirEndereco()
20     {
21         string textoEnd = "\nENDERECO\nCidade: " + this.Endereco.Cidade + "\nBairro: ";
22         textoEnd += this.Endereco.Bairro + "\nRua: " + this.Endereco.Rua + "\nNº casa: " + this.Endereco.Casa;
23         Console.WriteLine(textoEnd);
24     }
25
26     public override void exibirDonos()
27     {
28         Console.WriteLine("\nOs donos são:");
29         for (int i=0; i < this.Donos.Count; i++) {
30             Console.WriteLine(this.Donos[i].Nome);
31         }
32     }
33     public override void addDono(Dono dono) { this.Donos.Add(dono); }
34 }
```



C# - Associação

```
1 // Associação
2 class Dono {
3     public string Nome { get; set; }
4     public List<Casa> Casas { get; set; }
5
6     public Dono(string nome) {
7         this.Casas = new List<Casa>();
8         this.Nome = nome;
9     }
10
11    public void addCasa(Casa casa) { this.Casas.Add(casa); }
12 }
```

```
1 public class Program
2 {
3     public static void Main()
4     {
5         Dono dono1 = new Dono("Joao");
6         Dono dono2 = new Dono("Lari");
7         Dono dono3 = new Dono("Mario");
8
9         Endereco enderecol = new Endereco("Manaus", "Parque Dez", "Av. Darcy Vargas", 1200);
10
11        CasaBuilder planta = new PlantaCasa(enderecol);
12
13        Casa casa = planta.Casa;
14
15        casa.addDono(dono1);
16        casa.addDono(dono2);
17        casa.addDono(dono3);
18
19        casa.exibirDonos();
20    }
21 }
```

```
1 // Classe
2 class Casa : ACasa
3 {
4
5     public override List<Dono> Donos { get; set; }
6
7     public Casa() {
8         Donos = new List<Dono>();
9     }
10
11    public Casa(Endereco endereco) {
12        Donos = new List<Dono>();
13    }
14
15    public override void exibirDonos() {
16        Console.WriteLine("\nOs donos são:");
17        for (int i=0; i < this.Donos.Count; i++) {
18            Console.WriteLine(this.Donos[i].Nome);
19        }
20    }
21
22    public override void addDono(Dono dono) { this.Donos.Add(dono); }
23 }
```



C#

Agregação

```
1 // Agregação
2 // Um endereço existe mesmo sem ser vinculado a uma pessoa
3 public class Endereco
4 {
5     public string Cidade { get; set; }
6     public string Bairro { get; set; }
7     public string Rua { get; set; }
8     public int Casa { get; set; }
9
10    public Endereco(string cidade, string bairro, string rua, int casa)
11    {
12        this.Cidade = cidade;
13        this.Bairro = bairro;
14        this.Rua = rua;
15        this.Casa = casa;
16    }
17 }
```

```
1 // Classe
2 class Casa : ACasa
3 {
4
5     public override List<Quarto> Quartos { get; set; }
6     public override List<Dono> Donos { get; set; }
7     public override Endereco Endereco { get; set; }
8
9     public Casa() {
10         Quartos = new List<Quarto>();
11         Donos = new List<Dono>();
12     }
13     public Casa(Endereco endereco) {
14         Quartos = new List<Quarto>();
15         Donos = new List<Dono>();
16         this.Endereco = endereco;
17     }
18
19     public override void exibirEndereco()
20     {
21         string textoEnd = "\nENDERECO\nCidade: " + this.Endereco.Cidade + "\nBairro: ";
22         textoEnd += this.Endereco.Bairro + "\nRua: " + this.Endereco.Rua + "\nNº casa: " + this.Endereco.Casa;
23         Console.WriteLine(textoEnd);
24     }
25
26     public override void exibirDonos() {
27         Console.WriteLine("\nOs donos são:");
28         for (int i=0; i < this.Donos.Count; i++) {
29             Console.WriteLine(this.Donos[i].Nome);
30         }
31     }
32
33     public override void addDono(Dono dono) { this.Donos.Add(dono); }
34 }
```



C#

Composição

```
1 // Composição
2 // Uma parede e uma porta não fazem sentido sem estarem em um quarto
3 class Parede
4 {
5     public string Cor { get; set; }
6
7     public Parede(string cor)
8     {
9         this.Cor = cor;
10    }
11 }
12
13 class Porta
14 {
15     Quarto QuartoAtual { get; set; }
16     Quarto QuartoDestino { get; set; }
17
18     public Porta(Quarto quartoAtual, Quarto quartoDestino)
19     {
20         this.QuartoAtual = quartoAtual;
21         this.QuartoDestino = quartoDestino;
22     }
23 }
```

```
1 class Quarto
2 {
3     public int Id { get; set; }
4     public Parede ParedeEsq { get; set; }
5     public Parede ParedeDir { get; set; }
6     public Parede ParedeFrente { get; set; }
7     public Parede ParedeAtras { get; set; }
8     public Porta Porta { get; set; }
9
10    public Quarto(int id, Parede paredeEsq, Parede paredeDir, Parede paredeFrente, Parede
11                  paredeAtras)
12    {
13        this.Id = id;
14        this.ParedeEsq = paredeEsq;
15        this.ParedeDir = paredeDir;
16        this.ParedeFrente = paredeFrente;
17        this.ParedeAtras = paredeAtras;
18    }
19 }
```



Python Classe, Objeto e Construtor

```
# Classe
class Endereco:
    # Construtor
    def __init__(self, cidade: str = "", bairro: str = "",
                 rua: str = "", casa: int = 0) -> None:
        self.cidade = cidade
        self.bairro = bairro
        self.rua = rua
        self.casa = casa
```

```
# Objeto
enderec01 = Endereco("Manaus", "Parque Dez",
                     "Av. Darcy Vargas", 1200)
```

— — —



Python

Herança

```
from abc import ABC, abstractmethod

# Herança
class Casa_Estrutura(ABC):
    def __init__(self, endereco) -> None:
        self.quartos = []
        self.endereco = endereco
        self.donos = []

    @abstractmethod
    def exibir_endereco():
        pass

    @abstractmethod
    def exibir_donos():
        pass
```

```
# Classe concreta
class Casa(Casa_Estrutura):
    def __init__(self, endereco: Endereco = Endereco()) -> None:
        super(Casa, self).__init__(endereco)

    def exibir_endereco(self):
        print(f'''\nO endereço dessa casa é:\n
            Cidade: {str(self.endereco.cidade)}\n
            Bairro: {str(self.endereco.bairro)}\n
            Rua: {str(self.endereco.rua)}\n
            N° casa: {str(self.endereco.casa)}''')

    def add_dono(self, dono: Dono) -> None:
        dono.add_casa(self)

    def exibir_donos(self) -> None:
        print("\nOs donos dessa casa são:")
        for d in self.donos:
            print(d.nome)
```



Python Associação

```
# Associação
class Dono:
    def __init__(self, nome: str = "") -> None:
        self.nome = nome
        self.casas = []

    def add_casa(self, casa: Casa) -> None:
        self.casas.append(casa)
        casa.donos.append(self)
```

```
# Classe concreta
class Casa(Casa_Estrutura):
    def __init__(self, endereco: Endereco = Endereco()) -> None:
        super(Casa, self).__init__(endereco)

    def add_dono(self, dono: Dono) -> None:
        dono.add_casa(self)

    def exibir_donos(self) -> None:
        print("\nOs donos dessa casa são:")
        for d in self.donos:
            print(d.nome)
```

— — —



Python

Agregação

```
# Agregação
class Endereco:
    def __init__(self, cidade: str = "", bairro: str = "",
                 rua: str = "", casa: int = 0) -> None:
        self.cidade = cidade
        self.bairro = bairro
        self.rua = rua
        self.casa = casa
```

```
class Casa(Casa_Estrutura):
    def __init__(self, endereco: Endereco = Endereco()) -> None:
        super(Casa, self).__init__(endereco)

    def exibir_endereco(self):
        print(f'''\nO endereço dessa casa é:\n
            Cidade: {str(self.endereco.cidade)}\n
            Bairro: {str(self.endereco.bairro)}\n
            Rua: {str(self.endereco.rua)}\n
            N° casa: {str(self.endereco.casa)}'''')
```

— — —



Python Composição

```
# Composição
class Parede:
    def __init__(self, cor: str = "") -> None:
        self.cor = cor

class Porta:
    def __init__(self, id_quarto_atual: int = -1,
                 id_quarto_destino: int = -1) -> None:
        self.id_quarto_atual = id_quarto_atual
        self.id_quarto_destino = id_quarto_destino
```

```
class Quarto:
    def __init__(self, id: int = 0, parede_esq: Parede = Parede(),
                 parede_dir: Parede = Parede(),
                 parede_frente: Parede = Parede(),
                 parede_atras: Parede = Parede()) -> None:
        self.id = id
        self.parede_esq = parede_esq
        self.parede_dir = parede_dir
        self.parede_frente = parede_frente
        self.parede_atras = parede_atras
        self._portas = []

    @property
    def portas(self):
        return self._portas

    @portas.setter
    def portas(self, porta: Porta = Porta()):
        self._portas.append(porta)
```

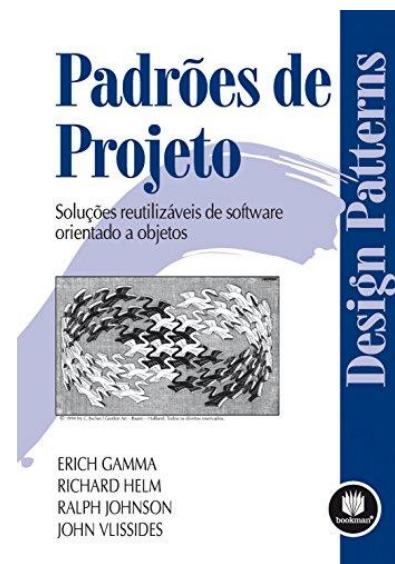


Tentando defender python: herança x composição

Desvantagens da Herança:

- Torna rígida em tempo de compilação, impossibilitando adaptação em tempo de compilação;
- O código da classe pai fica exposto a classe filha;
- Novas classes filhas podem exigir adaptação da classe pai;
- Mudanças da classe pai requerem reescrita em todas as filhas;
- As classes filhas devem implementar toda a classe pai, então se alguma característica não for necessária, deverá ser implementada mesmo assim;
- Dificulta a reutilização de classes filhas.

"Porque a herança expõe para uma subclasse os detalhes da implementação dos seus ancestrais, frequentemente é dito que 'a herança viola o encapsulamento' [sny86]". (Padrões de Projeto, p.35)





Tentando defender python: prefira composição a herança

*"Prefira a composição de objetos a herança de classes".
(Padrões de Projeto, p.35)*

DELEGAÇÃO:

- Um tipo mais específico de composição;
- Guarde uma instância da classe que seria a mãe quando precisar delegue a atividade a ela.

```
class Edificacao_Estrutura():
    def __init__(self, endereco) -> None:
        self.quartos = []
        self.endereco = endereco
        self.donos = []

    def exibir_endereco(self):
        print(f'''\nO endereço dessa casa é:\n
            Cidade: {str(self.endereco.cidade)}\n
            Bairro: {str(self.endereco.bairro)}\n
            Rua: {str(self.endereco.rua)}\n
            N° casa: {str(self.endereco.casa)}''')

    def add_dono(self, dono: Dono) -> None:
        dono.add_casa(self)

    def exibir_donos(self):
        print("\nOs donos dessa casa são:")
        for d in self.donos:
            print(d.nome)
```

```
class Casa:
    def __init__(self, endereco: Endereco = Endereco()) -> None:
        # Guardando instancia
        self.__acasa = Casa_Estrutura(endereco=endereco)

    def exibir_endereco(self):
        self.__acasa.exibir_endereco()

    def add_dono(self, dono: Dono):
        self.__acasa.add_dono(dono=dono)

    def exibir_donos(self) -> None:
        self.__acasa.exibir_donos()

    def get_quartos(self):
        return self.__acasa.quartos

    def add_quarto(self, quarto: Quarto = Quarto()):
        self.__acasa.quartos.append(quarto)
```



Tentando defender python: prefira interface a herança

"I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible". (artigo: why extends is evil)

```
class Casa_Builder(ABC):
    @abstractmethod
    def criar_parede():
        pass

    @abstractmethod
    def criar_quarto():
        pass

    @abstractmethod
    def criar_porta():
        pass

    @abstractmethod
    def inserir_endereco():
        pass
```

```
class Planta_Casa(Casa_Builder):
    def __init__(self, endereco: Endereco) -> None:
        self._casa = Casa()

        self.inserir_endereco(endereco)
        self.criar_quarto(1)
        self.criar_quarto(2)
        self.criar_porta(1, 2)

    @property
    def casa(self):
        return self._casa

    def inserir_endereco(self, endereco: Endereco):
        self._casa.endereco = endereco

    def criar_parede(self, cor: str):
        return Parede(cor=cor)

    def criar_quarto(self, id: int):
        p1 = self.criar_parede("branca")
        p2 = self.criar_parede("branca")
        p3 = self.criar_parede("branca")
        p4 = self.criar_parede("azul")

        self._casa.add_quarto(Quarto(parede_esq=p1, parede_dir=p2,
                                     parede_frente=p3, parede_atras=p4))

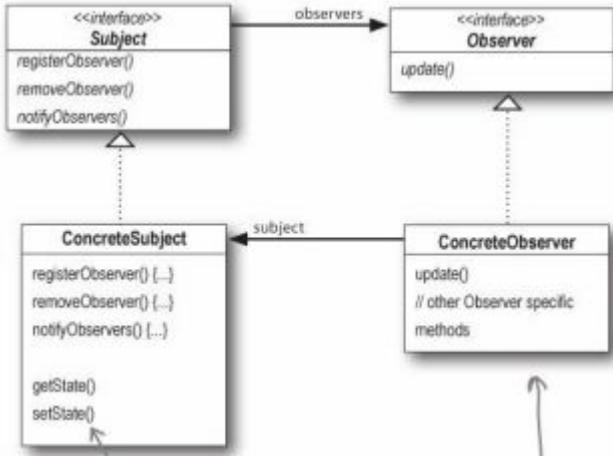
    def criar_porta(self, id_quarto1: int, id_quarto2: int):
        id_quarto_atual = -1
        id_quarto_destino = -1

        for i in range(len(self._casa.get_quartos())):
            if (self._casa.get_quartos()[i].id == id_quarto1):
                id_quarto_atual = self._casa.get_quartos()[i].id
            elif (self._casa.get_quartos()[i].id == id_quarto2):
                id_quarto_destino = self._casa.get_quartos()[i].id

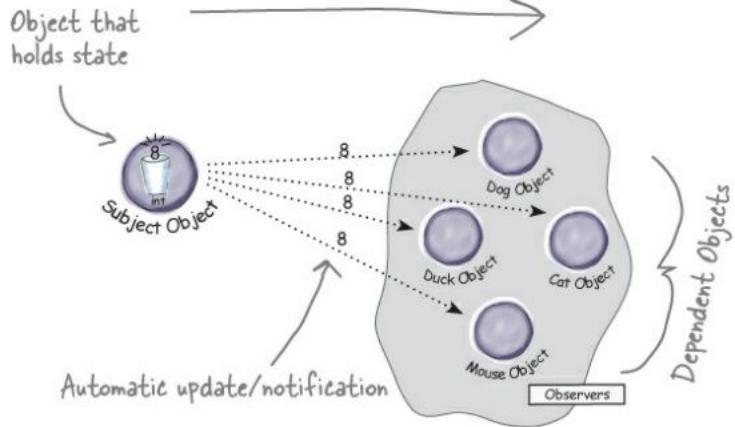
        id_not_null = (id_quarto_atual != -1 and id_quarto_destino != -1)
        id_not_equals = (id_quarto_atual != id_quarto_destino)
        if (id_not_null and id_not_equals):
            porta = Porta(id_quarto_atual, id_quarto_destino)
            self._casa.quartos[id_quarto_atual].portas = porta
```

Padrão de projeto Observer

Para que serve e como funciona



ONE-TO-MANY RELATIONSHIP





Ruby

Declarando as interfaces
Subject e Observer

```
1 module Interface
2   def method(name)
3     define_method(name) { |*args|
4       raise "interface method #{name} not implemented"
5     }
6   end
7 end
8
9 module Subject
10  extend Interface
11  method :registerObserver
12  method :removeObserver
13  method :notifyObservers
14 end
15
16 module Observer
17  extend Interface
18  method :update
19 end
```



Classe WeatherData

Atributos:

- `temperature` -> int
- `humidity` -> int
- `pressure` -> float
- `observers` -> `Observer[]`

```
1 class WeatherData
2   include Subject
3
4   # Declarando atributo
5   # de classe (getter e setter)
6   def temperature
7     @temperature
8   end
9
10  def temperature=( value )
11    @temperature = value
12  end
13
14  #... Outros Atributos
15  # humidity, pressure & observers
16
17  # constructor
18  def initialize
19    @observers = []
20  end
```



Classe WeatherData

Métodos:

- registerObserver(obs: Observer)
- removeObserver(obs: Observer)
- notifyObservers()
- measurementsChanged()
- setMeasurements(temperature, humidity, pressure)

```
22 def registerObserver(obs)
23   @observers << obs
24 end
25
26 def removeObserver(obs)
27   @observers.delete(obs)
28 end
29
30 def notifyObservers()
31   @observers.each { |obs| obs.update(@temperature,
32                                         @humidity,
33                                         @pressure) }
34 end
35
36 def measurementsChanged()
37   notifyObservers()
38 end
39
40 def setMeasurements(temperature, humidity, pressure)
41   @temperature = temperature
42   @humidity = humidity
43   @pressure = pressure
44   measurementsChanged()
45 end
46 end
47
```



Classe CurrentConditionsDisplay

Atributos:

- temperature -> int
- humidity -> int
- weatherData -> WeatherData

```
1 module DisplayElement
2   extend Interface
3   method :display
4 end
5
6 class CurrentConditionsDisplay
7   include Observer, DisplayElement
8
9   # Declaração de atributos
10  # temperature, humidity & weatherData
11
12  # constructor
13  def initialize(weatherData)
14    @weatherData = weatherData
15    @weatherData.registerObserver(self)
16  end
17
18  def update(temperature, humidity, pressure)
19    @temperature = temperature
20    @humidity = humidity
21    display()
22  end
23
24  def display()
25    puts 'Current conditions %d F degrees and %d humidity' %
26      [@temperature, @humidity]
27  end
28 end
```



Classe CurrentConditionsDisplay

Métodos:

- `update(temperature, humidity, pressure)`
- `display()`

```
1 module DisplayElement
2   extend Interface
3   method :display
4 end
5
6 class CurrentConditionsDisplay
7   include Observer, DisplayElement
8
9   # Declaração de atributos
10  # temperature, humidity & weatherData
11
12 # constructor
13 def initialize(weatherData)
14   @weatherData = weatherData
15   @weatherData.registerObserver(self)
16 end
17
18 def update(temperature, humidity, pressure)
19   @temperature = temperature
20   @humidity = humidity
21   display()
22 end
23
24 def display()
25   puts 'Current conditions %d F degrees and %d humidity' %
26     [@temperature, @humidity]
27 end
28 end
```



Java

```
● ● ●  
package headfirst.designpatterns.observer.weather;  
  
import java.util.*;  
  
public class WeatherData implements Subject {  
    private List<Observer> observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList<Observer>();  
    }  
  
    public void registerObserver(Observer o) {  
        observers.add(o);  
    }  
  
    public void removeObserver(Observer o) {  
        observers.remove(o);  
    }  
  
    public void notifyObservers() {  
        for (Observer observer : observers) {  
            observer.update(temperature, humidity, pressure);  
        }  
    }  
  
    public void measurementsChanged() {  
        notifyObservers();  
    }  
  
    public void setMeasurements(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        measurementsChanged();  
    }  
  
    public float getTemperature() {  
        return temperature;  
    }  
  
    public float getHumidity() {  
        return humidity;  
    }  
  
    public float getPressure() {  
        return pressure;  
    }  
}
```



Declarando as interface
Subject e classe abstrata Observer

```
interface Subject
{
    void RegisterObserver(Observer observer);
    void RemoveObserver(Observer observer);
    void NotifyObservers();
}
```

```
public abstract class Observer
{
    public abstract void Update(float temperature, float pressure, float humidity);
}
```



Classe WeatherData

Atributos:

- temperature -> int
- humidity -> int
- pressure -> float
- observers -> Observer[]

```
public class WeatherData : Subject
{
    private List<Observer> observers;
    private float temperature;
    private float pressure;
    private float humidity;

    public WeatherData(float temperature, float pressure, float humidity)
    {
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        this.observers = new List<Observer>();
    }

    public void RegisterObserver(Observer observer)
    {
        this.observers.Add(observer);
    }

    public void RemoveObserver (Observer observer)
    {
        this.observers.Remove(observer);
    }

    public void NotifyObservers()
    {
        foreach (Observer observer in this.observers)
        {
            observer.Update(this.temperature, this.pressure, this.humidity);
        }
    }

    public void SetMeasurements(float temperature, float pressure, float humidity)
    {
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        this.NotifyObservers();
    }
}
```



Classe CurrentConditionsDisplay

Métodos:

- update(temperature, humidity, pressure)
- display()

```
0 references
public class CurrentConditionsDisplay : Observer , DisplayElement
{
    2 references
    private float temperature;
    2 references
    private float humidity;
    2 references
    private WeatherData weatherData;

    0 references
    public CurrentConditionsDisplay(WeatherData weatherData)
    {
        this.weatherData = weatherData;
        this.weatherData.RegisterObserver(this);
    }

    0 references
    public override void Update(float temperature, float pressure, float humidity)
    {
        this.temperature = temperature;
        this.humidity = humidity;
        this.Display();
    }

    1 reference
    public void Display()
    {
        Console.WriteLine("Current Conditions: " + this.temperature.ToString() +
        "C degrees and " + this.humidity.ToString() + "% humidity");
    }
}
```



Classe Weather_Data

Implementa a superclasse
abstrata Subject

```
class Subject(ABC):

    @abstractmethod
    def register_observer(self, observer: Observer) -> None:
        pass

class Weather_Data(Subject):
    observers: List[Observer] = []
    temperature: float = None
    humidity: float = None
    pressure: float = None

    def __init__(self):
        self.observers.clear()

    def register_observer(self, observer: Observer) -> None:
        self.observers.append(observer)

    def remove_observer(self, observer: Observer) -> None:
        self.observers.remove(observer)

    def notify_observers(self) -> None:
        for observer in self.observers:
            observer.update(self.temperature, self.humidity, self.pressure)

    def measurements_changed(self) -> None:
        self.notify_observers()

    def set_measurements(self, temperature, humidity, pressure) -> None:
        self.temperature = temperature
        self.humidity = humidity
        self.pressure = pressure
        self.measurements_changed()
```



Classe Current_Conditions _Display

Implementa as superclasses
abstratas Observer e
Display_Element

```
class Observer(ABC):
    @abstractmethod
    def update(self, temperature, humidity, pressure) -> None:
        pass

class Display_Element(ABC):
    @abstractmethod
    def display() -> None:
        pass

class Current_Conditions_Display(Observer, Display_Element):
    temperature: float = None
    humidity: float = None
    weather_data: Subject = None

    def __init__(self, weather_data: Subject):
        self.weather_data = weather_data
        weather_data.register_observer(self)

    def update(self, temperature, humidity, pressure) -> None:
        self.temperature = temperature
        self.humidity = humidity
        self.display()

    def display(self) -> None:
        print(f'Current conditions: {self.temperature}F \
degrees and {self.humidity}% humidity')
```