Paradigma Orientado a Objetos

Herança e Composição

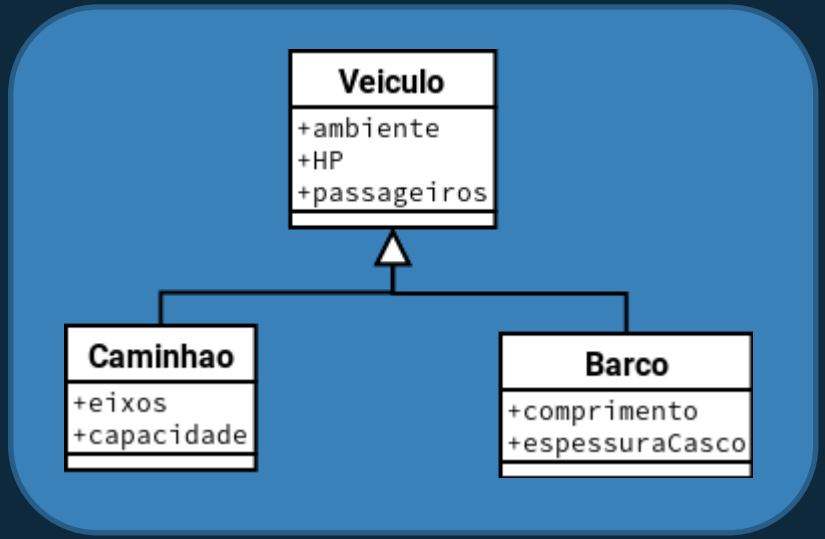
ERICK GALANI MAZIERO
erick.maziero@ufla.br

Departamento de Ciências da Computação
Universidade Federal da Lavras

UNIVERSIDADE FEDERAL DE LAVRAS

Herança em OO

- Mecanismo que permite que características comuns a diversas classes sejam organizadas em uma classe base e que, a partir dessa, outras possam ser criadas, herdando a classe base.



Herança em OO

- A classe derivada (ou subclasse) mantém as características herdadas e acrescenta o que for de sua exclusividade.

Veja o diagrama de classes do slide anterior.
Exemplo em Java

```
public class Barco extends Veiculo {
        private float comprimento;
        private float espessuraCasco;

    public Barco (String a, int hp, int p, float c, float e){
        super(a, hp, p);
        this.comprimento = c;
          this.espessuraCasco = e;
    }

}
```

Herança em OO

A classe derivada (ou subclasse) mantém as características herdadas e acrescenta o que for de sua exclusividade.

```
class Pessoa:
    def __init__(self, nome):
    self.nome = nome

class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        super().__init__(nome)
        self.med_id = med_id

class Medico(Pessoa):
    def __init__(self, nome,
id_func):
        super().__init__(nome)
        self.id_func = id_func
```

Herança em OO

Classe base. Contém características comuns a qualquer pessoa.

```
class Pessoa:
    def __init__(self, nome):
    self.nome = nome
```

Classe que herda a classe Pessoa. Adiciona atributos específico a um paciente

```
class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        super().__init__(nome)
        self.med_id = med_id
```

Classe que também herda a classe Pessoa. Adiciona atributos específico a um médico

```
class Medico(Pessoa):
    def __init__(self, nome, id_func):
        super().__init__(nome)
        self.id_func = id_func
```

Classe Abstrata

- Uma classe abstrata contém métodos abstratos, ou seja, que não têm implementação
- As classes que herdarem a classe abstrata são obrigados a realizar a implementação dos métodos abstratos da classe abstrata
- Uma classe abstrata, com métodos abstratos não pode ser diretamente instanciada

Classe Abstrata

Módulo nativo do Python para lidar com classes e métodos abstratos

Classe abstrata, ela herda a classe 'abc.ABC'.

Método abstrato, definido pelo *decorator* @abc.abstractmethod. As classes que herdarem Pessoa, deverão implementar o método definir_nome

```
import abc
class Pessoa(abc.ABC):
    @abc.abstractmethod
    def definir_nome(self, nome):
        pass


pessoa1 = Pessoa()    # erro: Can't instantiate abstract class Pessoa with abstract methods definir_nome
```

Classe Abstrata

Classe abstrata

Classe Paciente herda a classe abstrata Pessoa e implementa o método definir_nome

```
import abc
class Pessoa(abc.ABC):
    @abc.abstractmethod
    def definir_nome(self, nome):
        pass


class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        self.med_id = med_id
        self.definir_nome(nome)
    def definir_nome(self, nome):
        self.nome = nome
```

```
>> p1 = Paciente('João', 36348)
>> p1.nome
João
```

Duck Typing

- Estilo de codificação, em linguagens dinamicamente tipadas, em que define-se classes e métodos sem se importar com o tipo das variáveis.

Importa-se com o comportamento, não com o tipo

se anda como pato, nada como um pato e faz quack como um pato, então provavelmente é um pato

Duck Typing

- Por ser uma linguagem não tipada, ou seja, não se define o tipo das variáveis, os argumentos de métodos não são tipados e podem receber qualquer tipo de dados.
- Obviamente, as expressões com tais argumentos devem envolver operadores que consigam lidar com os valores fornecidos.
- Para se certificar que uma variável é de um tipo esperado, o Python fornece algumas funções úteis:
  - type()
  - isinstance()

`type()` e `isinstance()`

- `type()` recebe como parâmetro uma variável e retorna o tipo da mesma
- `isinstance()` recebe dois parâmetros: variável e tipo esperado. Retorna `True` se a variável é do tipo indicado e False, caso contrário

type() e isinstance()

```
class A:
    pass

>> a = A()

>> type(a)
<class '__main__.A'>

>> isinstance(a, A)
True
```

Herança múltipla

- Uma classe pode herdar de mais de uma classe seus atributos e métodos
- Java não suporta
- C++ e Python suportam herança múltipla

Herança múltipla

Considere a classe `Clock`

Implementa o método *tick*, que incrementa os segundos

```
class Clock():

    def __init__(self, hours, minutes, seconds):
        self.set_Clock(hours, minutes, seconds)

    def set_Clock(self, hours, minutes, seconds):
        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self.__minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between 0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self.__seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between 0 and 59!")

    def __str__(self):
        return "{0:02d}:{1:02d}:{2:02d}".format(self._hours,
                                                self.__minutes,
                                                self.__seconds)

    def tick(self):
        if self.__seconds == 59:
            self.__seconds = 0
            if self.__minutes == 59:
                self.__minutes = 0
                if self._hours == 23:
                    self._hours = 0
                else:
                    self._hours += 1
            else:
                self.__minutes += 1
        else:
            self.__seconds += 1
```

Herança múltipla

Considere a classe `Calendar`

```
class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    @staticmethod
    def leapyear(year):
        if not year % 4 == 0:
            return False
        elif not year % 100 == 0:
            return True
        elif not year % 400 == 0:
            return False
        else:
            return True

    def __init__(self, d, m, y):
        self.set_Calendar(d,m,y)

    def set_Calendar(self, d, m, y):
        self.__days = d
        self.__months = m
        self.__years = y

    def __str__(self):
        return "{0:02d}/{1:02d}/{2:4d}".format(self.__months,self.__days, self.__years)

    def advance(self):
        max_days = Calendar.months[self.__months-1]
        if self.__months == 2 and Calendar.leapyear(self.__years):
            max_days += 1
        if self.__days == max_days:
            self.__days= 1
            if self.__months == 12:
                self.__months = 1
                self.__years += 1
            else:
                self.__months += 1
        else:
            self.__days += 1
```

Implementa o método *advance*, que incrementa os dias

Herança múltipla

- Classe *CalendarClock*
  - Tem herança múltipla

Herança múltiplica das classes *Clock* e *Calendar*

Tem um método *tick*, que usa o *tick* da classe *Clock* e o método *advance* da classe *Calendar*

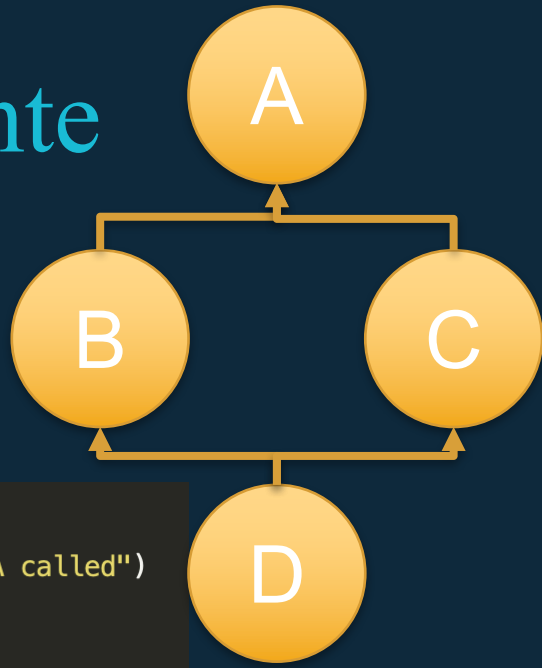
```
from clock import Clock
from calendar import Calendar


class CalendarClock(Clock, Calendar):
    def __init__(self,day, month, year, hour, minute, second):
        Clock.__init__(self,hour, minute, second)
        Calendar.__init__(self,day, month, year)


    def tick(self):
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)
```

Problema do Diamante



- O Problema do Diamante (devido à forma geométrica da ilustração ao lado) pode ocorrer na herança múltipla
- Considere as classes A, B, C e D, ao lado, o que acontece no seguinte trecho de código?

```
d = D()
d.m()
```

- Qual método m() será invocado, da classe A, B ou C?

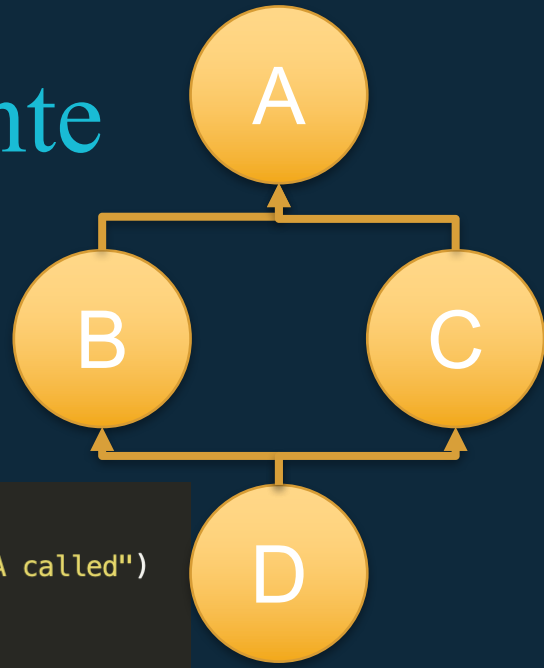
```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Problema do Diamante



- A resolução da ambiguidade depende da MRO (*Method Resolution Order*) de cada linguagem

- Leia em https://www.python.org/download/releases/2.3/mro/

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Padrões de Projeto de Software: GoF *(Gangue of Four)*

1994

- Padrões de Projeto de Software, conhecidos como GoF, traz soluções gerais para problemas frequentes no desenvolvimento de software

- São 24 padrões agrupados em 3 famílias:
  - Criacionais
  - Estruturais
  - Comportamentais

Padrões de Projeto de Software: GoF *(Gangue of Four)*

1994

- Os padrões de projeto de software são estudos na disciplina de engenharia de software.

- Mas, por estarem muito relacionado com o paradigma Orientado a Objetos, a seguir alguns exemplos são exemplificados em Python

Singleton

- Forma de definir classes que permitem a instanciação de um e apenas objeto
- Controlar criação de objetos

```
class Singleton:
    __instance = None
    @staticmethod
    def getInstance():
        if Singleton.__instance == None:
            return Singleton()
        return Singleton.__instance
    def __init__(self):
        if Singleton.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            Singleton.__instance = self
```

Builder

- Ajuda na construção de objetos complexos pelo uso de um objeto que usa uma abordagem algorítmica
- Constrói um objeto final através de um procedimento

```
class Director:
    __builder = None
    def setBuilder(self, builder):
        self.__builder = builder
    def getCar(self):
        car = Car()
        # First goes the body
        body = self.__builder.getBody()
        car.setBody(body)
        # Then engine
        engine = self.__builder.getEngine()
        car.setEngine(engine)
        # And four wheels
        i = 0
        while i < 4:
            wheel = self.__builder.getWheel()
            car.attachWheel(wheel)
            i += 1
        return car
```

Builder

```
class Car:
    def __init__(self):
        self.__wheels = list()
        self.__engine = None
        self.__body = None

    def setBody(self, body):
        self.__body = body

    def attachWheel(self, wheel):
        self.__wheels.append(wheel)

    def setEngine(self, engine):
        self.__engine = engine

    def specification(self):
        print "body: %s" % self.__body.shape
        print "engine horsepower: %d" % self.__engine.horsepower
        print "tire size: %d\'" % self.__wheels[0].size
```

```
class Builder:
        def getWheel(self): pass
        def getEngine(self): pass
        def getBody(self): pass

class JeepBuilder(Builder):

    def getWheel(self):
        wheel = Wheel()
        wheel.size = 22
        return wheel

    def getEngine(self):
        engine = Engine()
        engine.horsepower = 400
        return engine

    def getBody(self):
        body = Body()
        body.shape = "SUV"
        return body
```

Builder

```
jeepBuilder = JeepBuilder()

director = Director()

director.setBuilder(jeepBuilder)
jeep = director.getCar()
jeep.specification()
```

Factory

- Uma forma de criar objetos
- Usuário chama um método e passa uma string. Ele tem como retorno um objeto criado por um método *factory*

```
class Button(object):
    html = ""
    def get_html(self):
        return self.html

class Image(Button):
    html = "<img></img>"

class Input(Button):
    html = "<input></input>"

class Flash(Button):
    html = "<obj></obj>"

class ButtonFactory():
    def create_button(self, typ):
        targetclass = typ.capitalize()
        return globals()[targetclass]()
```

```
button_obj = ButtonFactory()
button = ['image', 'input', 'flash']
for b in button:
    print button_obj.create_button(b).get_html()
```

Associação *vs* Agregação *vs* Composição

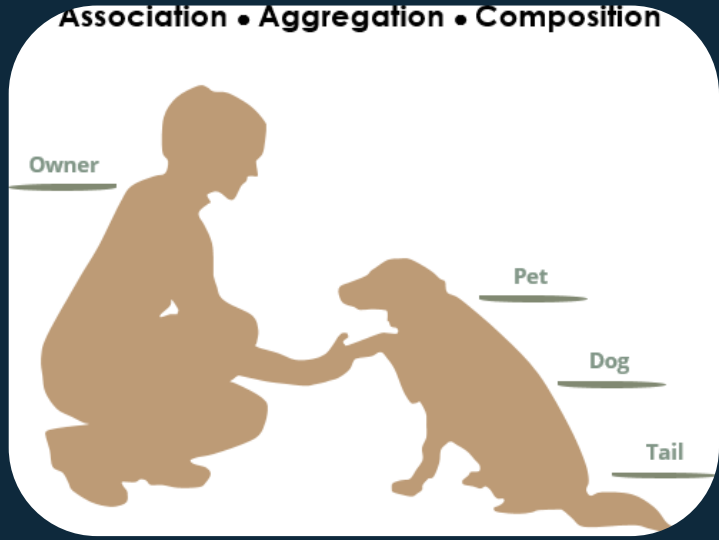
- Na orientação a objetos, um sistema é composto de várias classes (objetos) que têm algum tipo de relacionamento

- Alguns conceitos são importantes para entender e descrever os relacionamentos entre as classes, ou objetos, de um sistema

Associação *vs* Agregação *vs* Composição

Considere a figura ao lado. Cada rótulo indica uma classe. Então considere as possíveis formas de relacionamento entre essas classes

- **Associação**
  - owners alimentam pets e pets agradam owners
- **Agregação/Composição**
  - um tail é parte de dogs e também de cats
- **Herança**
  - dog e cat são tipos de pets



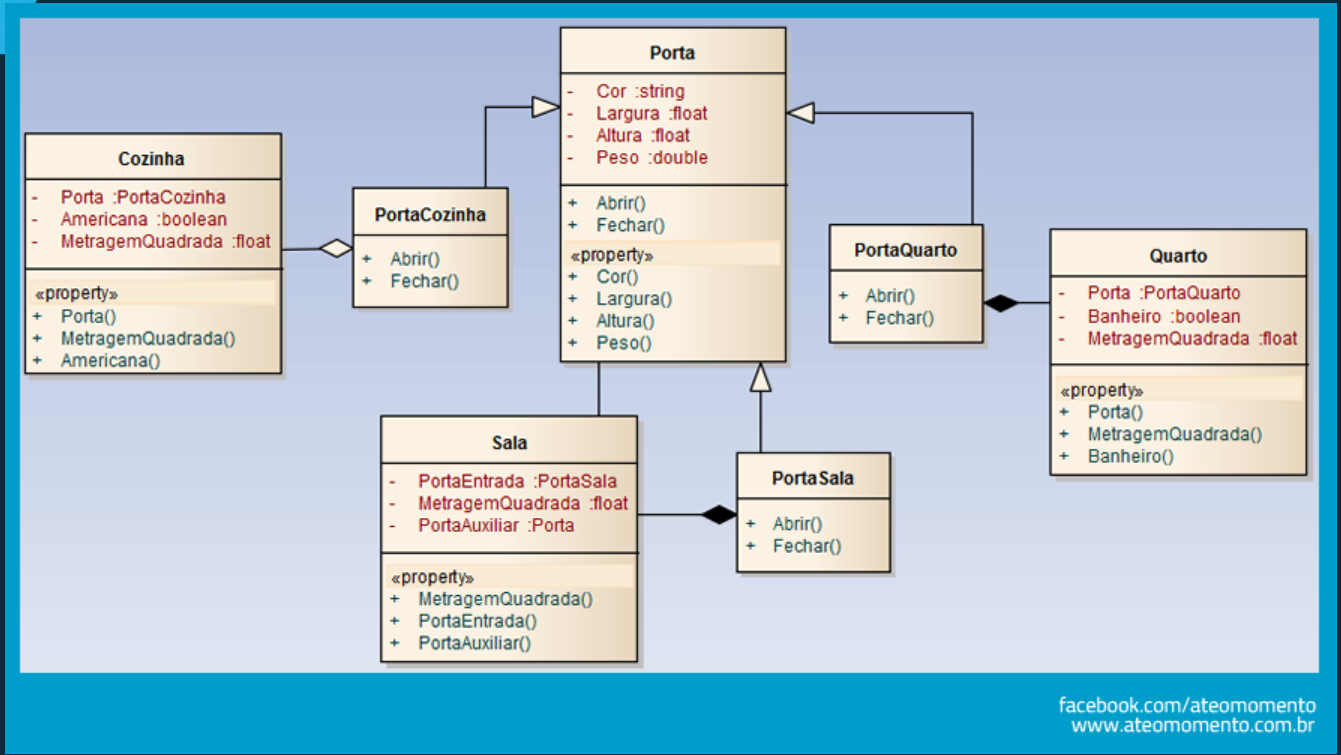
UML – *Unified Modeling Language*

- Uma forma gráfica de descrever os relacionamentos entre classes de um sistema está definido na UML (*Unified Modeling Language*) que é uma linguagem de notação (diagramas) para o projeto de sistemas

- Diagramas de classes
  - Apresenta as Classes, métodos, atributos indicando a arquitetura de um sistema orientado a objetos

- Considere o exemplo a seguir

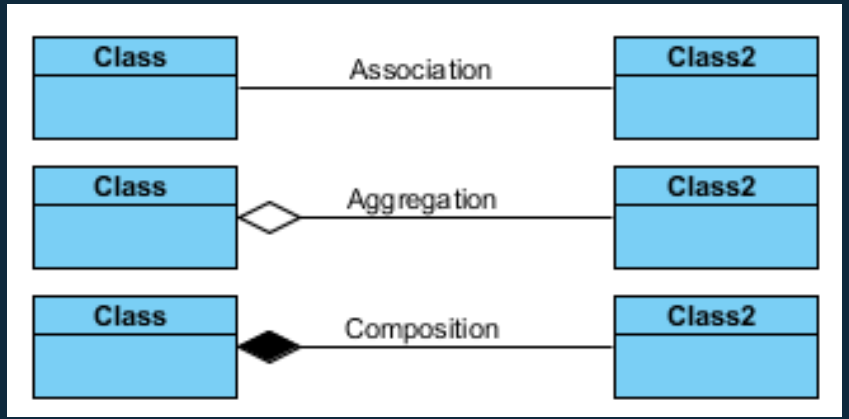
Diagrama de classes



Associação *vs* Agregação *vs* Composição

Observe a forma gráfica na figura ao lado

- **Associação**
  - Classes existem independentemente
- **Agregação/Composição**
  - Objetos dependem da existência de outros

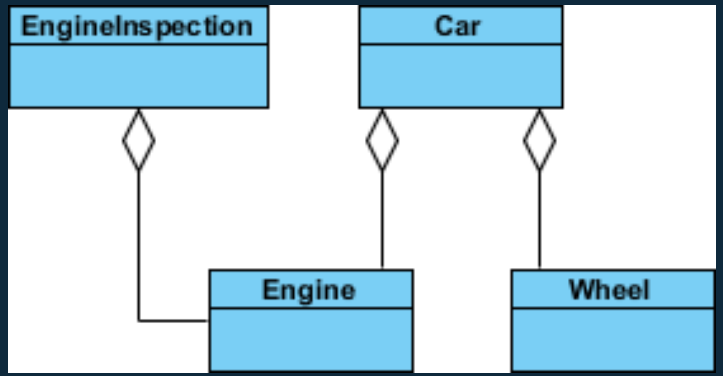


Associação

- Vínculo que ocorre entre classes
  - Associação binária
  - Inclusive associação própria (associação unária)
  - Outras associações (n-árias)

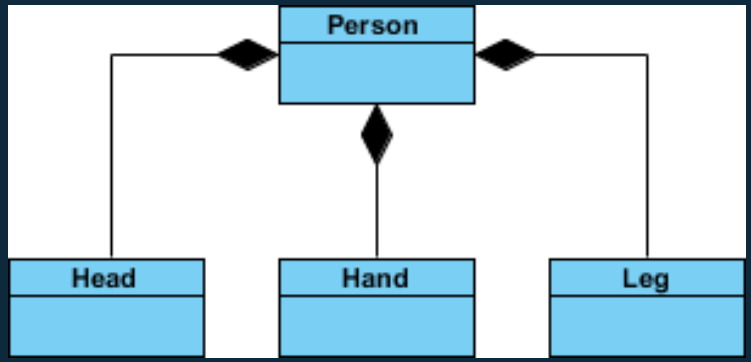
Agregação

- A classe Todo não é proprietária das classes Parte
- Agregação é utilizada para indicar que as classes Parte podem ter relações com outras classes Todo também



Composição

- Há uma dependência entre a classe Todo e as classes Parte
  - Quanto a classe Todo deixa de existir, as classes Parte também são destruídas



Acoplamento e Coesão

- É algo natural na confecção de um software
  - Mas em qual nível isso acontece entre as classes?
- Acoplamento fraco
  - Quando os componentes (classes, módulos, funcionalidades, tabelas, etc.) têm interdependência fraca
- Acoplamento forte
  - Componentes com forte interdependência

Acoplamento e Coesão

- São características importantes na descrição de um sistema desenvolvido orientado a objetos

- São medidas intra-classe (coesão) e interclasse (acoplamento)

Acoplamento

- É algo natural na confecção de um software
  - Mas em qual nível isso acontece entre as classes?
- Acoplamento fraco
  - Quando os componentes (classes, módulos, funcionalidades, tabelas, etc.) têm interdependência fraca
- Acoplamento forte
  - Componentes com forte interdependência

O que é melhor?

Acoplamento

- Baixo acoplamento, ou acoplamento fraco
  - Facilita manutenção do sistema
  - Ao arrumar um bug, outros *n* podem surgir

Coesão

- Medida intra elemento (classe, por exemplo)
  - Acoplamento: medida inter elementos
- Diz respeito ao conteúdo de cada elemento
  - A classe faz o que ela se propõe a fazer?
  - Há mistura de funcionalidades na classe?
  - Há mistura de ’objetos’ em uma classe?

Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.

Capítulo 12

Perkovic, L. *Introdução à Computação Usando Python - Um Foco no Desenvolvimento de Aplicações*. Editora LTC, 1. ed., 2016.

Capítulos 8 e 9.