4

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

4

Programação orientada a objetos

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo veremos finalmente o tema que dá nome à disciplina. Vamos aprender o que é uma classe e um objeto e qual a relação entre eles. Vamos situar a programação orientada a objetos em comparação com outros três paradigmas também bastante conhecidos: procedural, funcional e lógico. E por fim vamos introduzir os conceitos de abstração, encapsulamento, herança e polimorfismo, que juntos formam os quatro pilares da programação orientada a objetos.

4.1. Introdução

A programação orientada a objetos, que chamaremos de POO a partir de agora, é um paradigma de programação no qual o mundo real é modelado com base no conceito de objetos. Esse conceito aproxima a programação do mundo real, onde enxergamos as coisas à nossa volta de fato como objetos (televisão, mesa, carro, etc.) e isso facilita muito a resolução de diversos tipos de problemas.

Cada objeto possui características, uma TV por exemplo possui resolução e tamanho da tela, voltagem, cor, número de portas HDMI, e assim por diante. Em POO dizemos que estes são os ***atributos*** do objeto.

Os objetos possuem também ações, no exemplo da TV, podemos ligar, desligar, mudar de canal, ajustar o volume, alterar a entrada de vídeo, configurar o modo de exibição de cores, entre outros. Em POO dizemos que estes são os ***métodos*** do objeto.

Sendo assim, um programa em POO consiste na criação de diferentes objetos que irão interagir entre si para representar a situação que pretendemos modelar. Podemos dizer ainda que um objeto possui consciência de si mesmo e pode manipular

seus próprios dados. A TV do exemplo é capaz de alterar o valor do seu próprio volume, usamos aqui um dos métodos (ações) da TV para alterar um de seus atributos (características).

Existem basicamente duas abordagens para POO, baseada em classes e em protótipos. Atualmente a maioria das linguagens de programação orientadas a objetos é baseada em *classes*, termo que iremos falar bastante de agora até o final do curso. Uma classe é a abstração de um objeto, na qual definimos quais serão os atributos e métodos que os objetos de um mesmo tipo devem possuir. Podemos pensar na classe como um molde ou forma, que podemos usar para criar objetos.

Os conceitos aplicados atualmente em POO surgiram há muito tempo, com os termos "orientado" e "objetos" sendo usados neste contexto pela primeira vez no MIT[1], no final da década de 1960 (McCARTHY, 1960), em referência a elementos da linguagem LISP. Ao longo da década de 60, diversos estudos contribuíram para o desenvolvimento inicial desse paradigma, resultando no lançamento das primeiras linguagens orientadas a objetos, SIMULA e SMALLTALK, ainda no final da década.

Desde então, diversas novas linguagens foram criadas e ficaram conhecidas mundialmente, em especial a partir da década de 1990, impulsionadas também pela popularização das interfaces gráficas, que em geral se baseiam nas técnicas de POO. Dentre tais linguagens podemos citar Objective-C, C++, Java, C#, Delphi e Python.

Podemos dizer que praticamente todas as linguagens de programação se baseiam no conhecimento e nas experiências com uma ou mais linguagens anteriores, afinal, é assim que toda a ciência evolui, e com a ciência da computação não seria diferente. Portanto é muito comum observarmos semelhanças entre muitas dessas linguagens, que podem ser vistas no tratamento e manipulação dos dados em memória, no funcionamento do compilador ou interpretador da linguagem, nos recursos disponíveis para o programador, ou ainda na escolha das regras de sintaxe.

4.2. Paradigmas de programação

Um paradigma é uma forma de ver e interpretar a realidade, de acordo com o dicionário Michaelis (2021), um paradigma é "algo que serve de exemplo ou modelo; [um] padrão". Podemos dizer então que um paradigma de programação é um modelo que usamos para representar a realidade em nossos programas; é um conceito abstrato que nos diz como enxergar o mundo e suas relações e como traduzir isso para a programação. Podemos ainda pensar em um paradigma de programação como um estilo de programação.

Já uma linguagem de programação é algo concreto, que define um conjunto de regras de sintaxe com as quais podemos nos comunicar com o computador. Tais regras

[1] Instituto de Tecnologia do Massachusetts

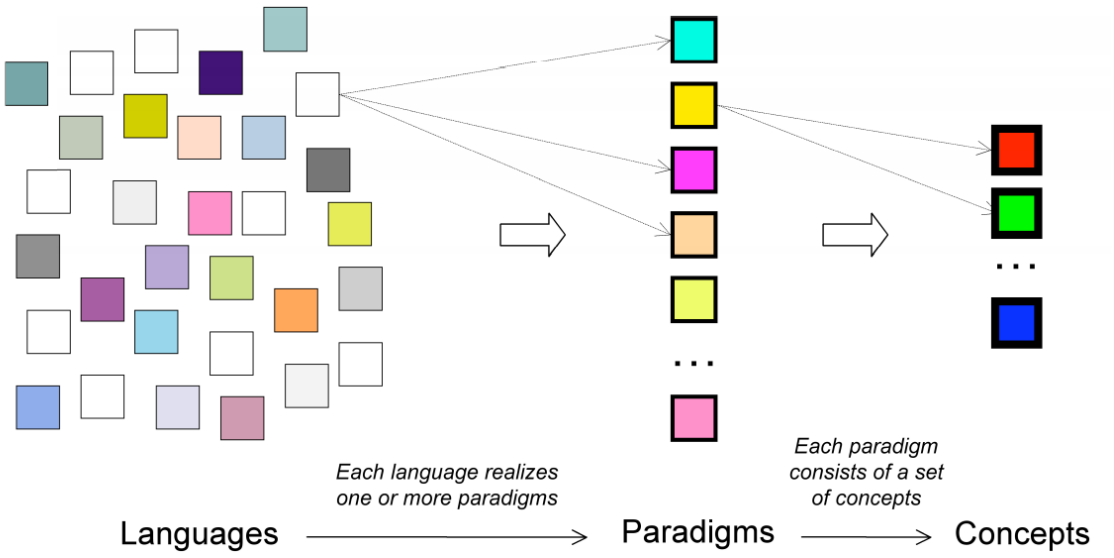
podem permitir o uso de um ou mais paradigmas, portanto uma linguagem não está limitada a um dado paradigma, e podemos classificá-las de acordo com quais paradigmas aceitam.

Podemos agrupá-los em dois grupos principais (COENEN, 1999):

1) *Imperativos*: o programa é constituído de uma sequência de instruções (ordens ou comandos) que dizem ao computador exatamente como manipular os dados. Fazem parte deste grupo os paradigmas procedural e orientado a objetos.
2) *Declarativos*: ao contrário do imperativo, o programa é composto de um conjunto de definições ou equações que descrevem o "que" o programa deve fazer, mas não o "como" fazer, que é delegado a implementação da linguagem. Fazem parte deste grupo os paradigmas funcional e lógico, entre outros.

É importante ressaltar que, assim como diferentes linguagens podem trabalhar com o mesmo paradigma, diferentes paradigmas podem compartilhar conceitos. A Figura 4.1 ilustra a relação entre estes três elementos: linguagens, paradigmas e conceitos.

Figura 4.1: Linguagens, Paradigmas e Conceitos



Fonte: ROY, 2009

Agora você pode estar se perguntando qual o melhor paradigma de programação, e a resposta mais uma vez depende do problema que você estiver resolvendo. A melhor forma de se programar é multiparadigma, pois diferentes problemas de programação precisam de diferentes conceitos para serem resolvidos de maneira elegante, e uma linguagem que suporte diferentes paradigmas dá ao programador a liberdade de aplicar o que for mais adequado à situação (ROY, 2009).

Por exemplo, a orientação a objetos é boa para problemas com uma grande quantidade de dados relacionais agrupados em uma estrutura hierárquica; para um problema com estruturas simbólicas complexas, o paradigma lógico é o mais adequado; já problemas com forte teor matemático, como análise de risco financeiro por exemplo, podem se beneficiar do uso do paradigma funcional.

4.2.1. Paradigmas Imperativos

Nos paradigmas imperativos, há um estado implícito que pode ser alterado através de comandos (HUDAK, 1989). Um programa escrito sob este modelo deve definir exatamente o que o computador deve fazer e em qual ordem, isto é, garantindo que o estado dos dados seja alterado de maneira determinística. Nós definimos os comandos que irão manipular os dados e o fluxo de execução destes comandos, podemos então, com base no estado atual, determinar o estado seguinte a partir dos comandos dados.

Em LP estudamos o paradigma procedural, no qual usamos estruturas de controle de fluxo para definir a ordem de execução dos comandos que queremos executar. Podemos ainda abstrair um conjunto de comandos ao agrupá-los em um procedimento (função), que pode então ser chamado como se fosse um comando único.

Na programação orientada a objetos, continuaremos a usar as mesmas estruturas de controle de fluxo para definir o que o computador deve fazer e em qual ordem. Então o que muda? Muda a forma como agrupamos estas instruções e os dados que elas podem manipular.

No paradigma procedural, aprendemos sobre o escopo global e local, nos quais podemos criar variáveis para salvar e manipular nossos dados. Mas agora, ao invés de agrupar as instruções em um procedimento, vamos agrupá-las em um objeto, que poderá conter dados internos e instruções sobre como manipular tais dados. Um programa então irá consistir na criação de diversos objetos que irão interagir entre si a partir da troca de mensagens.

4.2.2. Paradigmas Declarativos

Nos paradigmas declarativos, não há um estado implícito dos dados, e a programação é feita com base na avaliação de expressões ou termos (HUDAK, 1989). Os principais paradigmas declarativos são o lógico e o funcional.

O paradigma lógico é baseado na lógica formal, na qual definimos um conjunto de sentenças lógicas que expressam os fatos e regras pertinentes ao problema que queremos resolver, e a solução é deduzida a partir da aplicação de tais regras e fatos.

Já no paradigma funcional, as funções atuam como funções matemáticas puras, isto é, não alteram o estado do programa. Em outras palavras, não produzem nenhum

efeito colateral, retornando sempre o mesmo resultado se chamadas com os mesmos argumentos. Além disso, aspectos importantes do paradigma funcional são o tratamento de funções como objetos de primeira classe e a existência de funções de ordem superior. Ou seja, funções podem ser atribuídas a variáveis e passadas como argumento para outras funções, que podem também retornar uma nova função como valor de resposta.

Em um programa puramente funcional o foco está em declarar o que cada função deve fazer e, através da composição destas funções, chegar a solução de problemas. Em muitas linguagens de programação podemos aplicar conceitos da programação funcional à programação imperativa (procedural ou orientada a objetos).

4.3. Pilares de POO

Vamos trabalhar com a programação orientada a objetos baseada no conceito de classes, que são os blocos essenciais para a construção de um programa em POO. Há quatro ideias ou conceitos fundamentais da programação orientada a objetos:

- Abstração;
- Encapsulamento;
- Herança; e
- Polimorfismo.

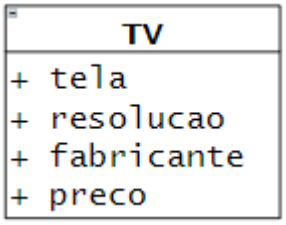
4.3.1. Abstração

O primeiro dos conceitos é provavelmente o mais importante de todos para a programação de maneira geral, pois ao representar algo do mundo real em um programa de computador, precisamos decidir o que iremos representar e o que iremos ignorar.

No contexto de POO, esse conceito ganha também o significado de generalização. Pense na TV que falamos na introdução, em qual TV você pensou? Uma TV de tubo? de LED? Grande? Pequena? FullHD? Smart? A TV que você pensou é um exemplar específico de uma TV, mas ela possui características em comum com todas as outras TVs, pois o conceito de TV é o mesmo e isso é o que chamamos de abstração.

Estamos interessados em uma generalização de TV, ou seja, a abstração em POO significa escolher as qualidades em comum de todas as TV que sejam relevantes para a situação que estamos querendo modelar. Pense que estamos escrevendo um programa para uma loja de eletrodomésticos e precisamos criar os anúncios das TVs que estarão disponíveis no site da loja. Para isso podemos criar uma classe TV que irá agrupar tudo que seja relevante nesta situação e a partir dessa classe, criar cada um dos objetos que irão representar os modelos específicos das TVs em estoque no momento.

Figura 4.2: Diagrama de classe simplificado

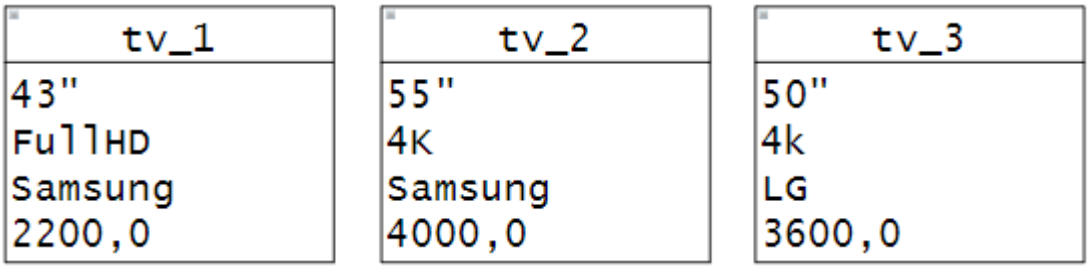


Fonte: do autor, 2021

Na Figura 4.2 vemos uma versão simplificada do diagrama de classe da UML[2], com o nome da classe e alguns dos possíveis atributos (características) que podemos escolher para uma TV. O sinal de + na frente de um atributo indica que ele é público, voltaremos neste assunto mais pra frente no curso.

E a partir dessa classe podemos criar quantos objetos de TV forem necessários. Cada objeto será único e terá seus próprios valores para cada atributo, como mostrado na Figura 4.3.

Figura 4.3: Diagrama de objetos simplificado



Fonte: do autor, 2021

4.3.2. Encapsulamento

O termo encapsulamento se refere a colocar algo no interior de um cápsula, em geral com os objetivos de manter junto e proteger. Em POO, isso pode fazer referência a própria classe em si, na qual agrupamos as propriedades e comportamentos que abstraímos do objeto real, e estamos modelando em uma unidade, um compartimento.

Mas mais do que isso, esse termo é usado para indicar a *ocultação de informações*. Podemos esconder partes da nossa classe do restante da aplicação, e isso serve a dois propósitos: proteger os dados do objeto de serem alterados por outra parte da aplicação que não seja o próprio objeto e esconder do restante da aplicação partes internas do funcionamento do objeto que podem sofrer alterações no futuro. Dessa

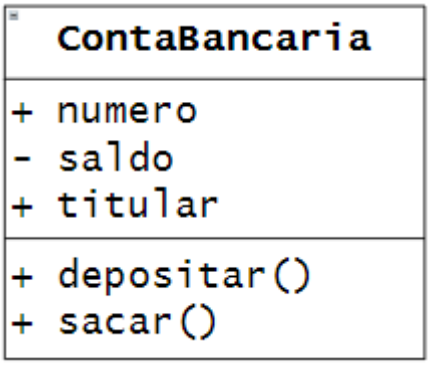
[2] UML é a Linguagem Unificada de Modelagem, da sigla em inglês *Unified Modelling Language*.

forma, caso algum detalhe interno da implementação seja alterado, isso não irá afetar a forma como os objetos são usados, contribuindo para uma boa manutenção do código.

Vamos pensar no exemplo de uma conta bancária, pense em quais atributos e quais métodos podemos abstrair para uma classe que represente as contas bancárias de um determinado banco.

A Figura 4.4 mostra uma possível modelagem para esta classe. Observe que não seria interessante ter o saldo da conta como um atributo público, que pudesse ser alterado por qualquer parte da aplicação, então para evitar isso, podemos esconder esse atributo internamente e deixar que a interação com o restante da aplicação se dê por meio de métodos públicos.

Figura 4.4: Exemplo de representação da classe para uma conta bancária



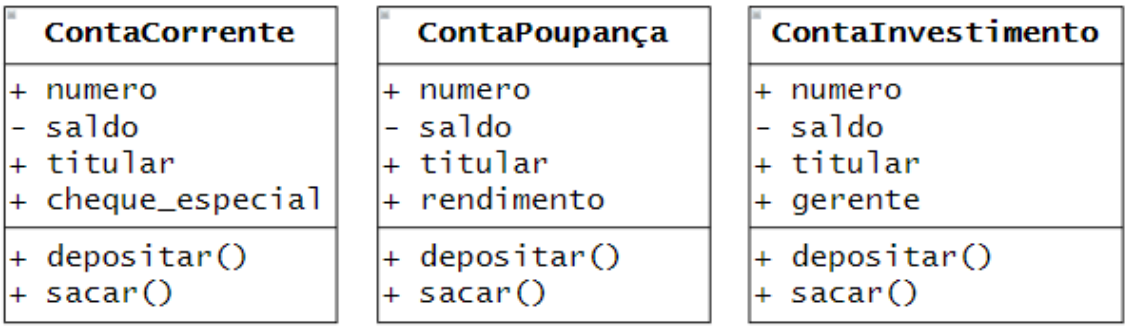
Fonte: do autor, 2021

No diagrama de classes, o sinal de menos indica que o atributo ou método é privado, ou seja, só está acessível ao próprio objeto, e não pode ser alterado por outra parte da aplicação. Neste exemplo as únicas formas de alterar o saldo da conta são através dos métodos sacar e depositar, e isso aumenta a segurança do funcionamento do nosso programa, pois podemos implementar restrições e verificações no interior desses métodos antes de fazer a alteração do valor do atributo.

4.3.3. Herança

A ideia de herança é uma das várias formas que podemos reutilizar código em POO. Digamos que no exemplo do banco precisamos agora criar outros tipos de contas, como uma conta corrente, poupança, investimento, etc. Poderíamos criar uma classe para cada tipo de conta, como mostra a Figura 4.5.

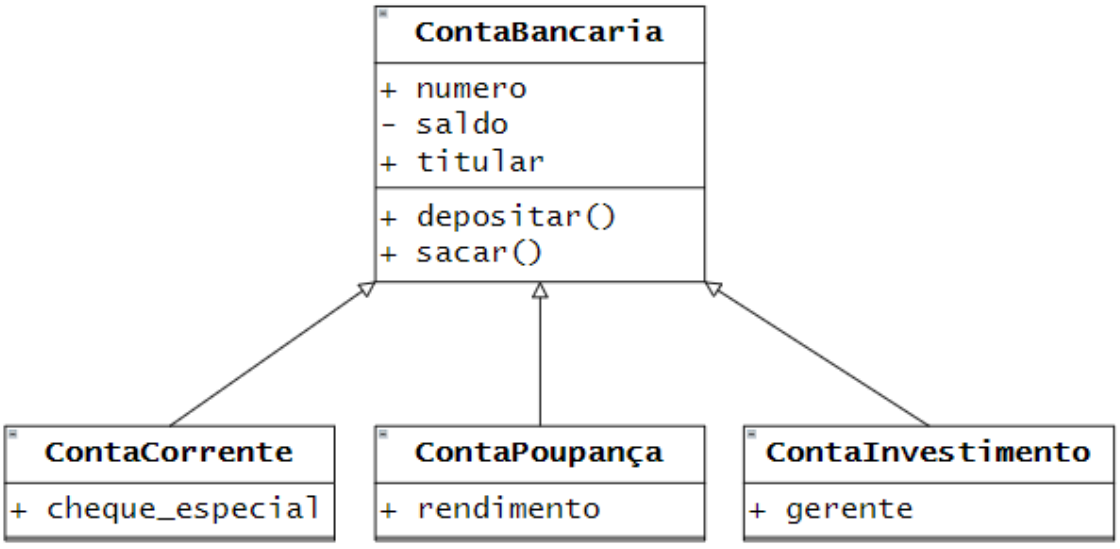
Figura 4.5: Representação das classes para os diferentes tipos de conta



Fonte: do autor, 2021

Observe no entanto que há várias características e comportamentos em comum entre estas classes, então a herança é uma forma de reaproveitarmos o código, criando uma classe nova a partir de outra classe já existente. Dizemos que a classe original é a superclasse ou classe mãe, e as classes derivadas são subclasses ou classes filhas. Veja a representação dessa relação na Figura 4.6.

Figura 4.6: Representação das classes da Figura 4.5 criadas com herança



Fonte: do autor, 2021

Com a herança, precisamos apenas adicionar os atributos ou métodos específicos de cada subclasse, todos os métodos e atributos em comum são herdados da classe mãe.

4.3.4. Polimorfismo

O polimorfismo é uma característica que aparece também em outros paradigmas de programação e pode ser dividido em dois tipos diferentes, sobrecarga e sobrescrita. Mas antes de falar sobre os dois tipos, vamos entender de onde vem esta palavra. Polimorfismo é uma palavra de origem grega:

- *Poly*: muito, numeroso, frequente;
- *Morph*: forma; e
- *Ismos*: processo, estado;

Ou seja, polimorfismo é uma propriedade daquilo que pode apresentar muitas formas ou aspectos, e em programação se refere a funções ou métodos com o mesmo nome, mas com comportamentos diferentes. Vejamos um exemplo que está presente em praticamente todas as linguagens de programação para entender melhor.

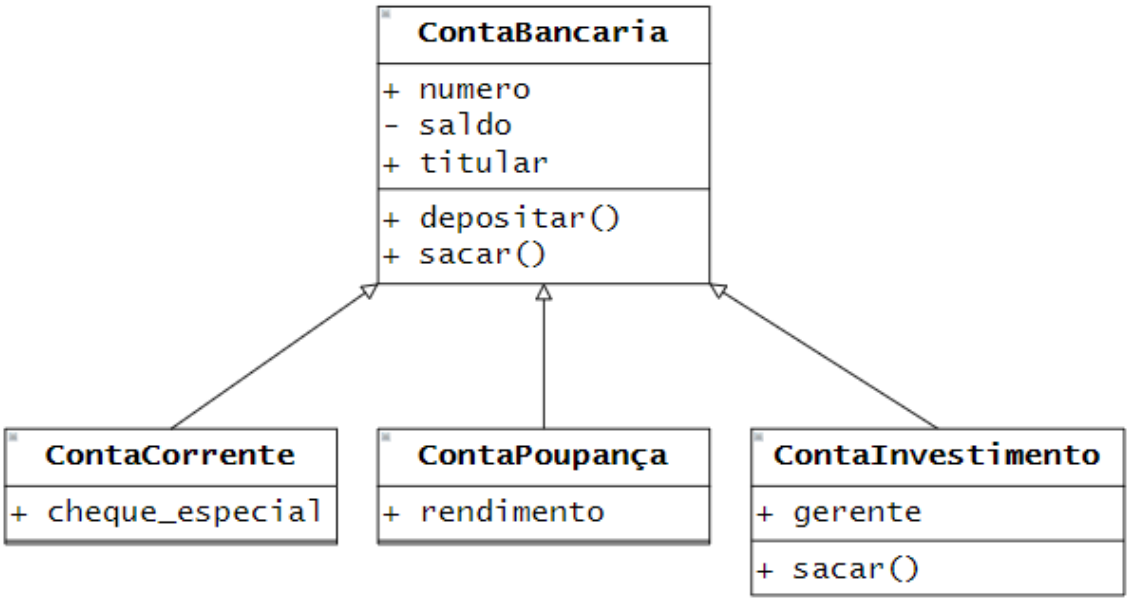
Em LP usamos o operador `+` para somar dois números e também para concatenar duas *strings*, duas listas ou duas tuplas. Em cada uma destas situações, a operação realizada é diferente, mas o operador é o mesmo. Dizemos então que este operador está sobrecarregado, isto é, carregado com mais de uma implementação, com mais de uma forma, daí o nome polimorfismo. A escolha de qual operação deve ser realizada é feita automaticamente em função dos operandos utilizados.

Agora vamos ver como isso se aplica às nossas classes do exemplo da conta bancária. No caso da conta de investimento, é comum haver um custo para sacar um valor investido, que em geral diminui em função do tempo de duração do investimento, além de eventuais cálculos de impostos ou taxas que podem ser cobrados. Com isso, não é interessante ter a conta investimento herdando o método *sacar* da classe mãe, pois ele estará incorreto.

No entanto, não adianta corrigir este método na classe mãe, pois isso também estaria incorreto, já que não é uma boa prática colocar a responsabilidade de um cálculo específico em uma classe genérica. Teríamos um método que só seria usado por uma das classes filha, mas todas as outras também o herdariam.

Uma possível solução seria criar esta classe do zero, sem a herança, mas com isso perderíamos a reutilização dos demais métodos e atributos que nos eram úteis, então a solução é mais simples do que pode parecer, basta ignorar a implementação da classe mãe e sobrescrever este método na classe filha, com a implementação das regras específicas para aquela classe. Veja na Figura 4.7 como ficaria tal representação.

Figura 4.7: Representação das classes da Figura 4.6 com polimorfismo do método sacar



Fonte: do autor, 2021

Agora, imagine que tenhamos 200 contas (objetos) diferentes em nosso programa, podemos seguramente chamar o método `sacar()` em qualquer uma delas e sabemos que todos os cálculos e verificações pertinentes serão realizados corretamente, seja ela uma conta corrente, poupança ou de investimento.

Com isso fechamos os principais conceitos de POO, nas próximas aulas, veremos como aplicar o que aprendemos aqui para criar classes e objetos em Python.

Bibliografia

COENEN, F. **Topics in information processing**. 1999. Disponível em: <https://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html>. Acesso em: 13 fev. 2021.

HUDAK, P. **Conception, evolution, and application of functional programming languages**. 1989. Disponível em: <http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf>. Acesso em: 13 fev. 2021.

MCCARTHY, J., et. al. **LISP I programmer's manual**. Artificial Intelligence Group, 1960. Disponível em: <http://history.siam.org/sup/Fox_1960_LISP.pdf>. Acesso em: 10 fev. 2021.

Michaelis. **Dicionário brasileiro da língua portuguesa**. 2021. Disponível em: <https://michaelis.uol.com.br/busca?r=0&f=0&t=0&palavra=paradigma>. Acesso em: 13 fev. 2021.

ROY, P. V. **Programming paradigms for dummies:** what every programmer should know. 2009. Disponível em: <https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf>. Acesso em: 13 fev. 2021.