ORIENTAÇÃO A OBJETOS

Sumário

O que é Programação Orientada a Objetos

A Programação Orientada a Objetos, ou POO, é um paradigma no qual uma linguagem de programação imita várias características do mundo real por meio de conceitos como Classe, Objetos, Atributos, Métodos, dentre outros. Como conceito resumido, dizemos que, num sistema implementado no paradigma POO há vários objetos que se comunicam entre si por meio da troca de mensagens.

Nos contextos de uso de programação para resolução de uma diversidade maior de problemas, se fez necessários criar um paradigma mais robusto e ao mesmo tempo mais abstrato de mapeamento desses cenários e situações a serem "implementadas" no mundo digital. Para programação Web principalmente essa colocação se torna ainda mais relevante uma vez que temos mais entidades e funcionalidades para compor um programa/plataforma.

Linguagens de programação que permitem POO

Para adotar esse paradigma é necessário usar uma linguagem de programação que permita seu uso. As linguagens de programação que permitem o uso de POO mais usadas no mundo são: **C#**, **C++**, **Java**, **Javascript**, **PHP**, **Python** e **Ruby** (em ordem alfabética). Para saber o grau de adoção das linguagens de programação em nível mundial, basta consultar os principais rankings internacionais, que são **TIOBE Index**[1], **IEEE Spectrum**[2], **RedMonk Rankings**[3] e **PYPL Index**[4].

Algumas linguagens implementam mais conceitos da POO que outras. Por exemplo, Java implementa mais conceitos de POO do que Python, aterrizando essa afirmação temos conceitos que serão detalhados posteriormente como polimorfismo e sobrescrita que em algumas linguagens é viável e em outras não. Outro fato interessante é que uma linguagem de programação pode implementar mais de um paradigma de programação. Javascript, por exemplo: Pode ser usado para programar em POO mas também permite usar o paradigma de Programação Funcional. Quando uma linguagem permite o uso de POO podemos dizer que se trata de uma linguagem orientada a objetos.

O que não é Orientação a Objetos

É importante deixar claro alguns conceitos que são muito confundidos com Orientação a Objetos que são **Programação Visual** e **Programação Baseada em Blocos**. Isso ocorre por confundirem os "objetos" nas interfaces dos programas com o termo "Objeto" da Orientação a Objetos. Para entender melhor, basta analisar um

[1] TIOBE Index - https://www.tiobe.com/tiobe-index/

[2] IEEE Spectrum - https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

[3] RedMonk Rankings - https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/

[4] PYPL (PopularitY of Programming Language) - http://pypl.github.io/PYPL.html

exemplo de Programação Visual na Figura 1, onde os **componentes** (e não “objetos”) são mais facilmente colocados, posicionados e configurados de forma visual na interface. Ou ainda, analisando um exemplo de Programação Baseada em Blocos na Figura 2, onde programa-se com o uso de **blocos** (e não “objetos”).

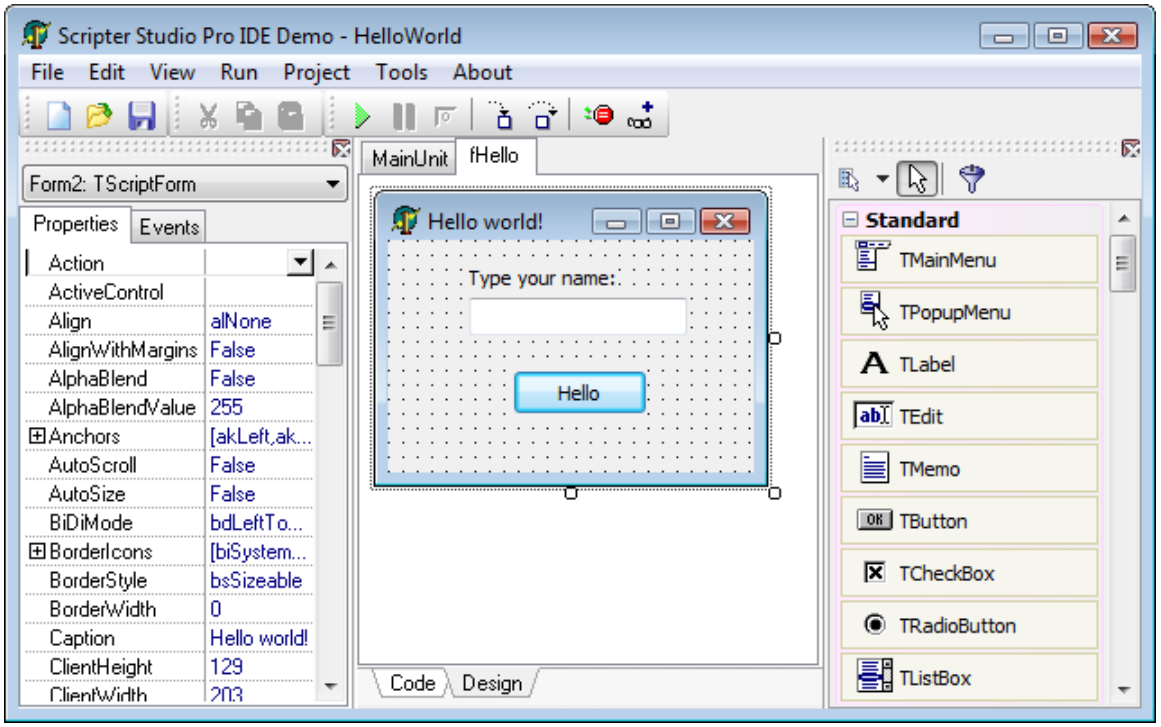

Figura 1: Tela da IDE Delphi, que possui recurso de Programação Visual
Fonte: https://www.tmssoftware.com/site/scriptstudiopro.asp

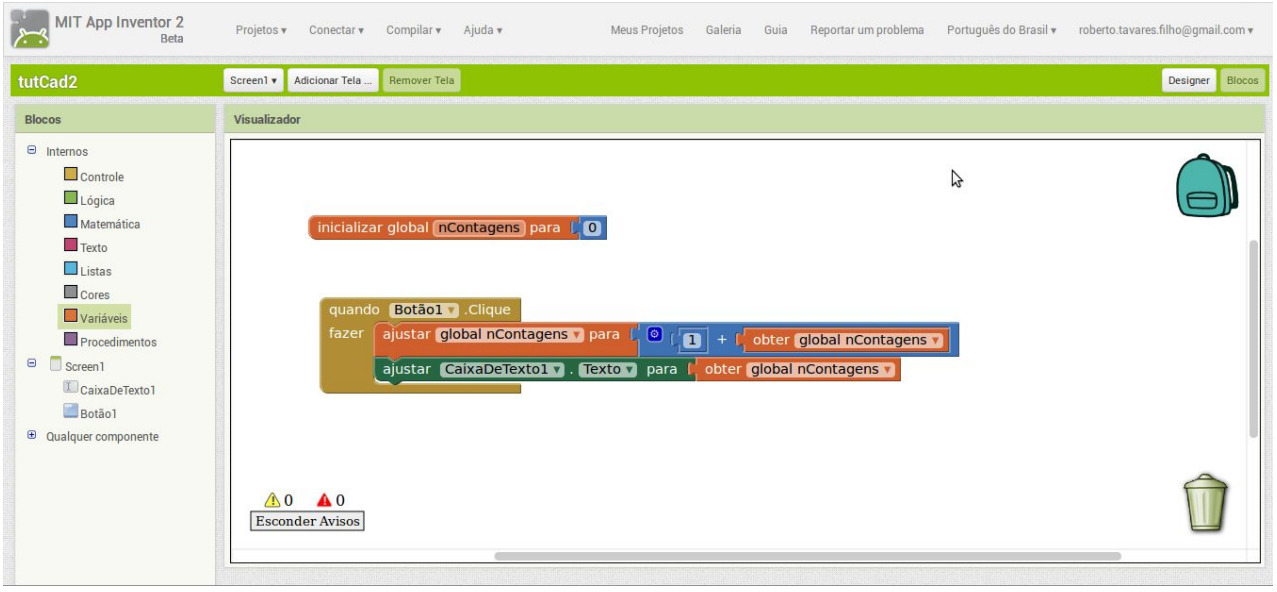

Figura 2: Tela do App Inventor, que possui recurso de Programação Baseada em Blocos
Fonte:https://cadernodelaboratorio.com.br/2016/12/15/desenvolvimento-android-com-o-appinventor-iii/

Principais conceitos de Orientação a Objetos

A seguir, os principais conceitos do paradigma da Orientação a Objetos.

Classe, atributos e métodos

Classes são definições (ou especificações) que abstraem um conjunto de objetos com características e comportamentos similares. As características chamados de **atributos** e os comportamentos chamamos de **métodos**. Na maioria das linguagens de programação orientadas a objetos, a prática mais comum é programar uma classe por arquivo de código fonte, ou seja, uma classe e seus atributos e métodos estão, necessariamente, em apenas um arquivo. É possível definir mais de uma classe por arquivo, embora não seja uma prática muito comum nem recomendada. A seguir, alguns exemplos de classes com seus atributos e métodos.

Classe "**carro**"

O que caracteriza um carro? Sua marca, seu modelo, seu ano, sua cilindrada, seu tipo de câmbio, sua quantidade de combustível no tanque, e por aí vai, certo? Então esses seriam seus **atributos**.

Quais os comportamentos de um carro? Ou seja, o que um carro é capaz de fazer, seja por si só, seja devido ao comando de outro personagem (o motorista, por exemplo)? Ele pode ser ligado, pode ser desligado, pode andar, pode acelerar, pode trocar de marcha etc. Então esses seriam seus **métodos**.

Classe "**triatleta**"

O que caracteriza um triatleta? O nome dele, seu peso, sua altura, quais os títulos que já conquistou, seu nível de excelência no ciclismo, na maratona e na natação etc. Então esses seriam seus **atributos**.

Quais os comportamentos de um triatleta? Ou seja, o que um triatleta sabe fazer, seja por si só, seja devido ao comando de outro personagem (um árbitro, um treinador, por exemplo)? Ele sabe pedalar, correr, nadar, aquecer, alongar etc. Então esses seriam seus **métodos**.

Classe "**conta corrente**"

O que caracteriza uma conta corrente? O nome de seu titular, os números de agência e conta, o saldo atual, a data de abertura etc. Então esses seriam seus **atributos**.

Quais os comportamentos de uma conta corrente? Ou seja, o que pode acontecer a uma conta, seja por si só, seja devido ao comando de outro personagem (o correntista, um gerente, uma rotina diária, por exemplo). Ela pode receber um depósito, pode ser usada para pagar uma conta, pode fazer ou receber uma transferência etc. Então esses seriam seus **métodos**.

Tente refletir no seu contexto de desenvolvimento de tecnologia quais seriam atores que deveriam estar representados nas plataformas. Pessoa? Estabelecimentos? Investimentos? Transação? Quando eu crio uma classe ou coloco algo como atributo de uma classe? Ou ainda quais são as classes do meu projeto? Essas são dúvidas comuns e representam a necessidade de falarmos de **Modelagem de projeto.** Não existem respostas certas e apenas uma única alternativa que funciona, mas existem boas práticas e maneiras de tirar máxima vantagem do paradigma orientado a objeto. Vamos trabalhar em práticas de modelagem no decorrer desta apostila.

Convenções para nomes de classes, atributos e métodos

Os **nomes de classes** seguem as mesmas regras de nomes de variáveis seguidas pela imensa maioria das linguagens de programação, tais como: Não podem começar com números, não podem ter espaço em branco, não podem ter caracteres usados em operações matemáticas, dentre outras. E a maioria das linguagens de programação que permitem orientação a objetos tem como convenção iniciar os nomes das classes com letra maiúscula e usar o padrão **CamelCase**.

Os **nomes dos atributos e métodos** também costumam seguir as mesmas regras de caracteres e o padrão **CamelCase**. A diferença é que, a maioria das linguagens começam com letra minúscula. Na linguagem **C#**, por exemplo, a convenção para nomes de métodos é iniciar com letra maiúscula.

Assim, os nomes das classes de exemplo citadas anteriormente seriam **Carro**, **Triatleta** e **ContaCorrente**. Seus nomes de alguns de seus atributos e métodos ficariam assim:

Atributos de **Carro**:

- marca
- modelo
- anoFabricacao

Métodos de **Carro**:

- ligar
- desligar
- trocarMarcha

Atributos de **Triatleta**:

- nome
- peso
- titulosConquistados

Métodos de **Triatleta**:

- pedalar
- correr
- nadar

Atributos de **ContaCorrente**:

- nomeTitular
- saldo

Métodos de **ContaCorrente**:

- depositar
- pagarConta
- obterSaldo

Objeto (ou Instância)

Já vimos que classe é a definição. Quando queremos que uma definição dessas seja concretizada em algo único, dizemos que **instanciamos a classe**, criando assim um **objeto** (ou **instância**) dessa classe.

É como se a classe fosse o projeto e o objeto (ou instância) fosse o produto criado a partir desse projeto. Supondo que Deus criou a ideia do que é um "ser humano", digamos que ele definiu a classe **SerHumano**. Então ele resolveu colocar em prática esse projeto e instanciou um objeto do tipo **SerHumano**, chamando ele de "**Adão**". Se esse momento tivesse sido fotografado, seria como o da famosa obra "A Criação de Adão", de Michelangelo (vide Figura 3).

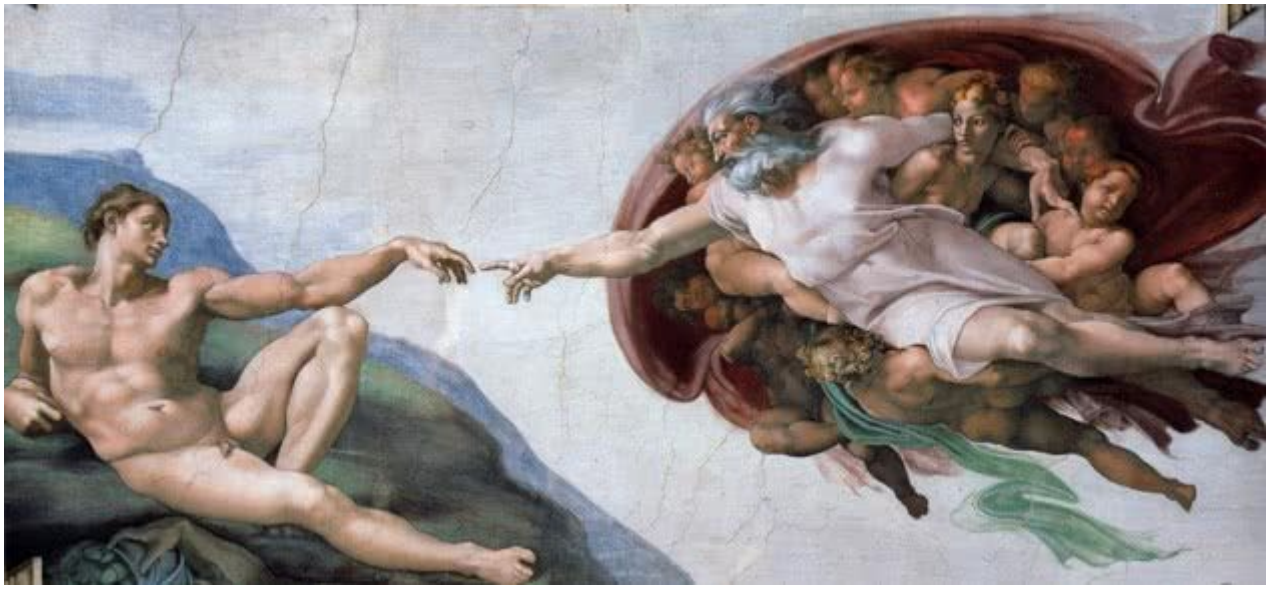

Figura 3: Famosa obra “A criação de Adão”. Deus finalizando a criação de Adão, dando-lhe alma.
Fonte: https://www.culturagenial.com/a-criacao-de-adao-michelangelo/

Usando um exemplo da engenharia aeronáutica: Um projeto de um caça F-15 seria sua classe **CacaF15** (vide Figura 4) e todos os jatos fabricados a partir desse projeto são os objetos do tipo CacaF15 (vide Figura 5).

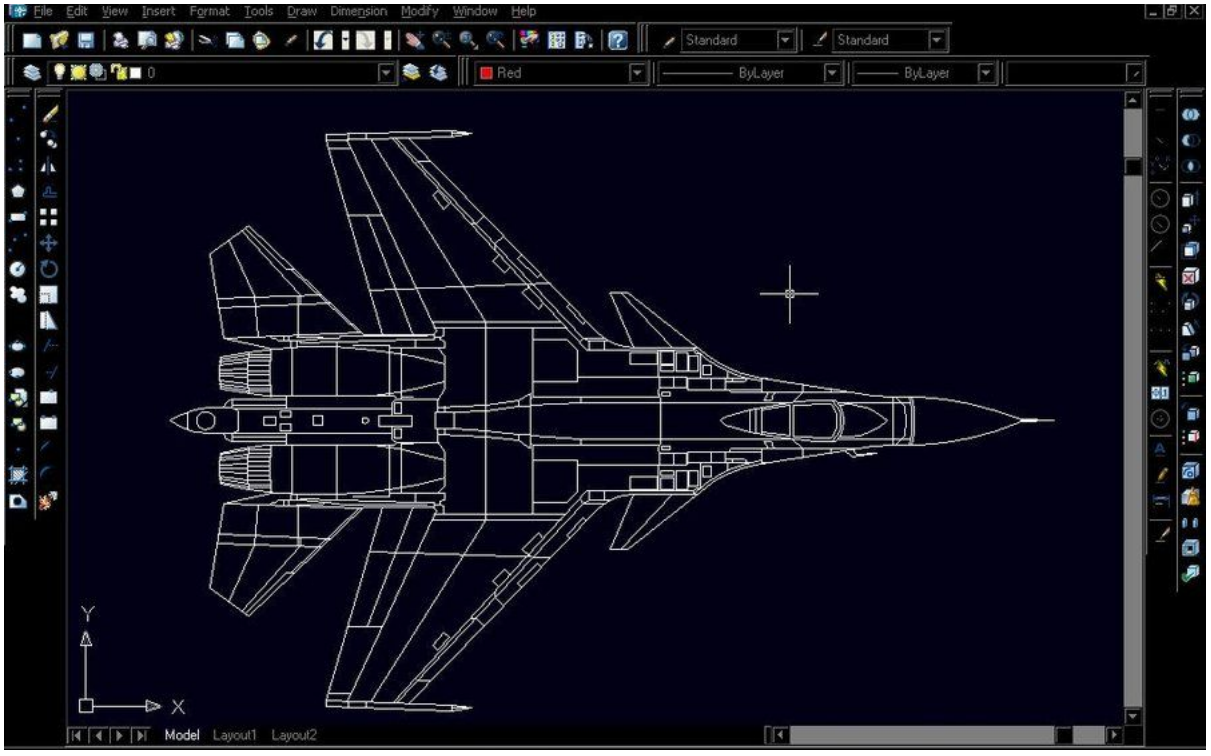

Figura 4: Projeto de um caça F-15. Seria o mais parecido com “classe” na engenharia.
Fonte: http://www.crowellphoto.com

Figura 5: Vários caças F-15. Poderíamos dizer que são vários objetos (instâncias) de CacaF15.
Fonte: https://www.businessinsider.com/air-force-upgrading-the-f-15-to-keep-its-edge-on-chinese-fighter-jets-2017-9

Objeto (ou instância) é um elemento que possui efetivamente as características (atributos) e executa as ações (métodos) descritas em sua Classe. **Cada objeto é único** e tem, digamos que, uma "vida própria". É como os caças F-15 fabricados em série: Nascem exatamente iguais, mas logo após saírem do pátio da fábrica podem passar a ter atributos com valores diferentes (ex: cor, quantidade de munição nas armas etc) e/ou terem métodos diferentes invocados (ex: em um é feito uma decolagem e em outro é solicitado o disparo de um míssil).

Quando pedimos a execução de um método em um objeto, dizemos que estamos **invocando** esse método. Exemplos: num objeto do tipo **ContaCorrente**, para que ele receba um valor em dinheiro, devemos invocar seu método **depositar**. Num objeto do tipo **Triatleta**, para que ele corra, invocamos seu método **correr**.

Detalhes sobre atributos

Tipo

Quando definimos os atributos de uma classe, temos que decidir qual é o **tipo** de cada um deles. Esse tipo indica que tipo de dado esse atributo pode armazenar. Os tipos vão variar conforme os tipos disponíveis da linguagem de programação usada. Todavia, existe uma certa convergência de nomes e finalidades de tipos entre as linguagens de programação modernas. Isso significa que existem tipos comuns a muitas linguagens, tais como:

- **String**, usado para armazenar valores alfanuméricos;
- **Integer**, para números inteiros;
- **Double**, para números reais;
- **Date**, para datas e/ou horas;

- **Boolean**, para valores lógicos (**true** ou **false**).

Por exemplo, na classe **Carro**, o atributo **marca** poderia ser do tipo **String** e o **anoFabricacao** poderia ser do tipo **Integer**.

Já na classe **ContaCorrente**, o atributo **nomeTitular** poderia ser do tipo **String** ou similar e o **saldo** poderia ser do tipo **Double** ou similar.

Nível de acesso (ou visibilidade) - básicos

Algum atributos podem ter seu valor alterado e recuperado de forma direta e outros não. Para ajustar o quão direto é um acesso a um atributo, definimos seu **nível de acesso** (ou, segundo alguns autores, sua "**visibilidade**").

Existem vários níveis de acesso, porém alguns deles só serão compreensíveis após estudarmos outros conceitos. Por agora, vamos conhecer apenas 2: **private** e **public**.

Quando um atributo é **private**, somente o código fonte da **própria classe** pode obter e alterar seu valor. No mundo real, esse é o nível de acesso mais usado para atributos.

Quando um atributo é **public**, tanto o código fonte da **própria classe** quanto os de outras classes podem obter e alterar seu valor. No mundo real, é muito pouco comum usar esse nível de acesso em atributos.

Para entender a diferença, vamos supor que na classe **ContaCorrente**...

- O **nomeTitular** foi definido como **público**, portanto qualquer código, pode recuperar e alterar o valor no **nomeTitular** de qualquer objeto do tipo **ContaCorrente**;
- O **saldo** foi definido como **privado**, portanto se algum código possui um objeto do tipo **ContaCorrente**, não é possível recuperar nem alterar o valor de **saldo** diretamente. Para alterar o **saldo**, só indiretamente, usando métodos como **depositar** ou **pagarConta**.

Detalhes sobre métodos

Tipo de retorno

Ao invocar um método pode-se esperar que ele devolva algum **valor**. Nesse caso dizemos que ele é um método **com retorno**. Quando um método possui retorno, ele deve ser de algum tipo suportado pela linguagem de programação em uso. Então tudo que já foi dito sobre **tipos** no tópico sobre atributos é igualmente válido aqui.

Um exemplo de método com retorno seria o **obterSaldo** de nossa classe **ContaCorrente**. Como o nome sugere, ao ser invocado, ele devolve o valor do saldo da conta para quem o invocou.

Outros tipos de métodos apenas são invocados sem que se espere que sua execução devolva algo. Esses métodos são conhecidos como métodos **sem retorno**. Na maioria das linguagens de programação, o termo usado para definir um método sem retorno é **void**.

Um exemplo de método sem retorno seria o **ligar** de nossa classe **Carro**. Apenas pede-se para o carro ligar, sem se esperar nenhum valor.

Importante: É claro que qualquer método pode ter problemas durante sua execução. Isso não significa que todo método deve retornar um valor (ex: mensagem de erro ou status de execução). A questão de erro em tempo de execução deve ser tratada com um recurso chamado **tratamento de exceções**, disponível em todas as linguagens de programação modernas, mas esse assunto é tema de tópicos futuros.

Nível de acesso (ou visibilidade) - básicos

O conceito aqui é muito parecido com o de nível de acesso de atributos. Um método pode ter vários níveis de acesso, dentre os que podemos estudar agora também são **private** e **public**.

Quando um método de uma classe é **public**, ele pode ser invocado de qualquer objeto do tipo dessa classe. Um exemplo de método candidato a ser **public** seria o método **ligar** da classe **Carro**. Quem manipula um carro, deve poder ligar ele, certo?

Quando um método de uma classe é **private**, ele só pode ser invocado no código da **própria classe**. Isso ocorre quando queremos isolar um “mini programa” dentro da classe para termos reaproveitamento e componentização dentro de uma classe mas não queremos que esse método seja visível, muito menos invocado de um objeto da classe. Imagine que precisamos criar um método chamado **ligarMotorArranque** na classe **Carro**. No mundo real, não há como ligarmos diretamente o motor de arranque de um carro, certo? Esse motor é ativado quando pedimos para ligar o carro. Isso ocorre porque convencionou-se na engenharia automotiva que não é seguro que exista um mecanismo direto (público) que permita alguém ligar o motor de arranque. Logo, o exemplo de método candidato a ser **private** seria o **ligarMotorArranque** da classe **Carro**. Assim, não seria possível invocar diretamente ele de uma instância de **Carro**.

Argumentos

Assim como ações do mundo real, alguns métodos precisam de alguma informação para poderem funcionar. Por exemplo, ao realizar a troca de marcha num carro precisamos informar para qual marcha queremos ir. Ou para realizar um depósito

em uma conta corrente, precisamos informar o valor do depósito. Não é como pedir para desligar o carro. É só pedir para desligar, pois não é preciso informar mais nada.

Então, quando identificamos que um método precisa de uma informação para funcionar, dizemos que ele precisa de um ou mais **argumentos**. Os argumentos seguem as mesmas regras de nome e de tipos dos atributos. A seguir, alguns exemplos de métodos que citamos aqui refatorados para o uso de argumentos.

Classe **Carro**, método **trocarMarcha** - 1 argumento: **novaMarcha** (tipo **Integer**). Assim, quem invocar o método **trocarMarcha**, será obrigado a informar o valor de **novaMarcha**.

Classe **ContaCorrente**, método **depositar** - 1 argumento: **valorDeposito** (tipo **Double**). Assim, quem invocar o método **depositar**, será obrigado a informar o valor de **valorDeposito**.

Um método pode possuir quantos argumentos forem necessários (embora existam autores que recomendam o máximo de quatro argumentos). Poderíamos ter um novo método em **ContaCorrente** chamado **pagarConta**, que precise de várias informações para ser invocado. Ficaria assim:

Classe **ContaCorrente**, Método **pagarConta** - 3 argumentos: **codigoBarras** (tipo **String**)**, dataPagamento** (tipo **Date**)**, valorPago** (tipo **Double**). Assim, quem invocar o método **pagarConta**, será obrigado a informar o valor de **codigoBarras**, **dataPagamento** e **valorPago**.

Classes no Diagrama de Classe

Até agora vimos alguns conceitos elementares da orientação a objetos. Mas talvez tudo tenha ficado muito abstrato até agora. Ou talvez você possa ter se perguntado como documentar as classes com seus atributos e métodos e com toda aquela questão de níveis de acesso, tipos, retornos etc.

Felizmente, existe uma linguagem de modelagem de sistemas padronizada internacionalmente chamada **UML - Unified Modeling Language** (“Linguagem de Modelagem Unificada” em tradução livre), cuja versão atual é 2.5.1, datada de dezembro de 2017. Ela define uma série de diagramas para modelar sistemas desenvolvidos com o paradigma de orientação a objetos. E o que vai nos ajudar agora é o **Diagrama de Classe**.

Nesse diagrama, uma classe é definida em um retângulo de qualquer cor de borda, preenchimento e texto. Esse retângulo deve estar dividido em 3 retângulos. O **nome da classe** fica no **retângulo superior**, seus **atributos** ficam no **retângulo central** e os **métodos** ficam no **retângulo inferior**. Vejamos como ficariam os diagramas de classe das classes de exemplo que descrevemos até agora. O diagrama da classe **Carro** está na Figura 6, o da classe **Triatleta** na Figura 7 e o da **ContaCorrente** na Figura 8.

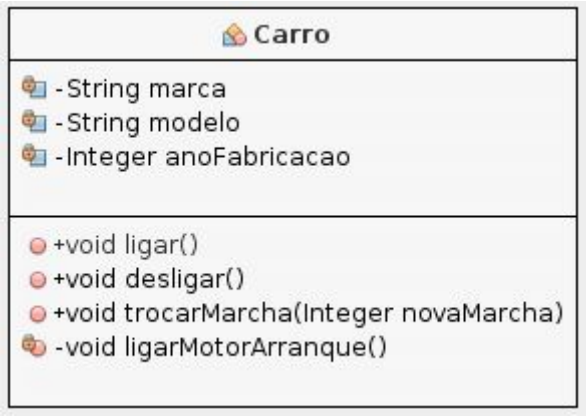

Figura 6: Diagrama da classe “Carro”.

Figura 7: Diagrama da classe “Tiratleta”.

Figura 8: Diagrama da classe “ContaCorrente”.

Note nos diagramas que há símbolos de **+** ou **-** antes dos atributos e métodos. Eles indicam o nível de acesso destes.

(+) Indica que um atributo ou método é **public**.

(-) Indica que um atributo ou método é **private**.

Logo após os modificadores de acesso, temos o tipo do atributo ou método. Logo após os métodos temos sempre um par de parênteses, o que é padrão na UML para indicar que são métodos. Caso existam argumentos, estes ficam dentro desse par

de parênteses. Cada argumento é descrito primeiro com seu tipo e depois com seu nome. No caso de mais de um argumento, são separados por vírgula.

Antes dos nome da classe e dos modificadores de acesso dos atributos e métodos há alguns ícones. Eles são **opcionais** na UML e servem para ajudar a descrever o membro que ele precede. Um ícone composto por um quadrado, um círculo e um losango é usado para indicar uma classe. Um ícone de um quadrado indica um atributo. Um ícone de um círculo indica um método. A presença de um cadeado sobre o ícone indica que o atributo ou método é **privado**.

Para praticar a criação de diagramas de classes, há uma ótima opção gratuita online, que depende apenas de conexão com internet e um browser atualizado. É o site **http://draw.io**.

Com os conceitos apresentados até o momento, estamos prontos para discutir tópicos de modelagem e arquitetura de programas orientados a objetos. Nas sessões seguintes vamos discutir princípios e boas práticas para isso.

Encapsulamento

Encapsulamento ocorre quando temos um atributos privado que só pode ser acessado, seja para leitura seja para escrita, por meio de um ou mais métodos públicos. Nós já fizemos isso aqui. Foi no caso do atributo **saldo** da classe **ContaCorrente**. Podemos dizer que ele está **encapsulado** pois é privado e pode ter seu valor lido por um método público (**obterSaldo**) e alteramos por outros métodos públicos (**depositar** e **pagarConta**).

Usamos esse recurso para proteger atributos de leituras e/ou alterações indesejadas. No saldo, por exemplo, é bem mais seguro só permitir que seja alterado por operações de conta corrente do que diretamente, não concorda?

Muitas linguagens modernas convergiram para um padrão de encapsulamento chamado **Get/Set**. Nele, os atributos privados podem ser lidos por um método público (um **getter**) e alterados por outro método público (um **setter**). No caso no Java, os getters são métodos que, literalmente, começam com “get” e terminam com o nome do atributo enquanto que os setters começam com “set” e terminam com o nome do atributo. Se fossemos encapsular o atributo **nomeTitular** da classe **ContaCorrente**, por exemplo, ele passaria a ser privado e teríamos 2 novos métodos públicos **getNomeTitular** (retorna uma String e sem argumentos) e **setNomeTitular** (sem retorno e recebe uma String como único argumento). O diagrama dessa classe ficaria como o da Figura 9.

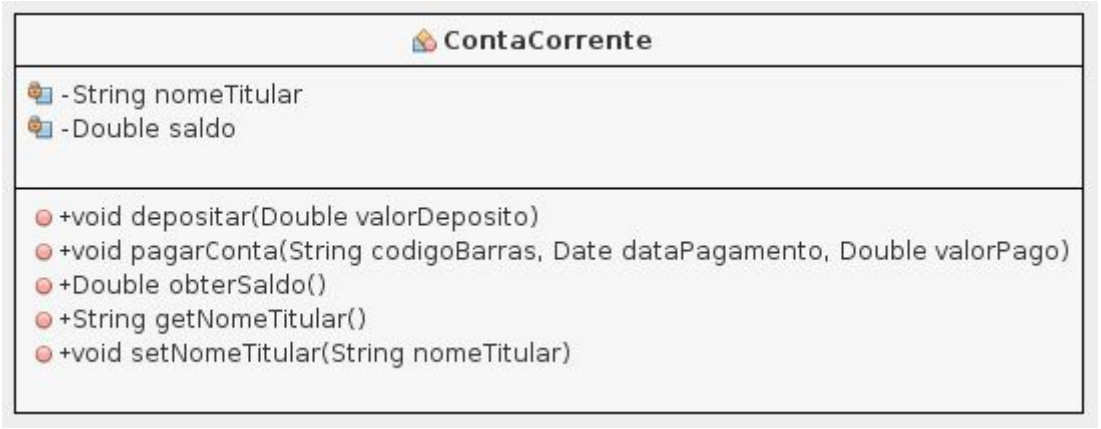

Figura 9: Diagrama da classe “ContaCorrente” com o atributo “nomeTitular” encapsulado no padrão “get/set”.

Uma outra forma de implementar o encapsulamento de atributos privados é promover a determinação de seu valor no **momento em que um objeto é criado**. Para isso, criamos uma coisa chamada **construtor**. Um construtor parece com um método, mas: não possui nenhum retorno (não é que retorna void, não possui nenhum retorno configurado mesmo) e seu nome é exatamente o nome da classe. Uma classe pode ter quantos construtores forem necessários. No mais, é como um método na questão dos parênteses e argumentos. Inclusive um construtor pode ser privado também (embora seja muito raro isso ser necessário). Poderíamos, por exemplo encapsular os atributos de **ContaCorrente** por meio de construtores públicos: um que só recebe o **nomeTitular** e outro que recebe valores dos 2 atributos. Assim, o diagrama da classe ficaria como o da Figura 10.

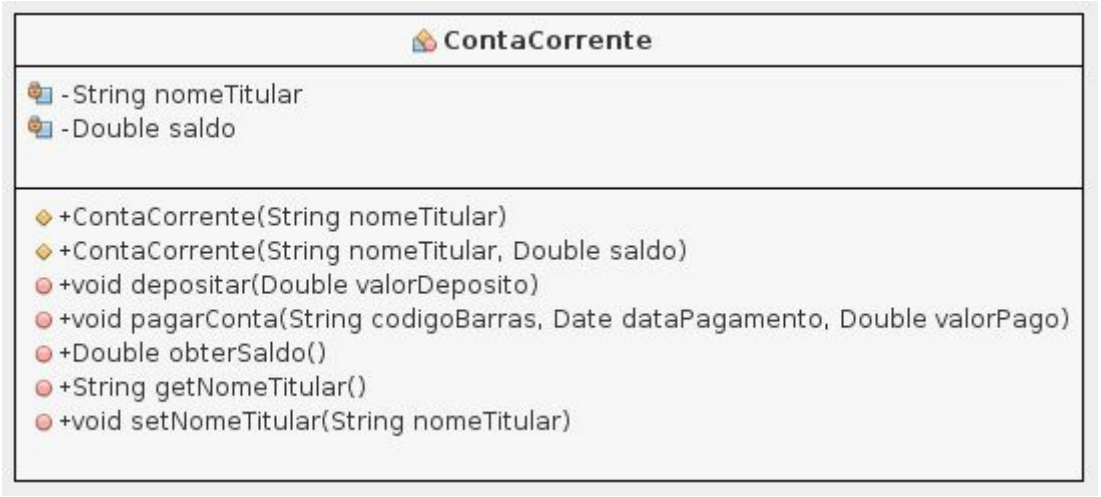

Figura 10: Diagrama da classe “ContaCorrente” dois construtores públicos, acima dos métodos.

Note que os construtores, acima dos métodos, estão marcados como públicos com o mesmo sinal **(+)** dos demais membros públicos da classe. E seu ícone é um losango.

Atividades práticas sugeridas

1. Atividade para verificar aprendizado de **modelagem** e **diagrama de classes**:

Marina prepara diversos exercícios para suas filhas. Elas estão na primeira e segunda série e querem ser astronautas. Ela gostaria de informatizar esses exercícios, para gerar testes aleatórios.

Cada teste gerado deve ser guardado, acompanhado de suas questões com a indicação da sua data de geração. Na geração de um teste, é preciso informar o número de questões desejadas e a qual disciplina pertence o teste.

Para cada disciplina, cadastra-se: uma lista de questões objetivas, identificando de que bimestre é cada questão e a que matéria pertence. O gabarito também é cadastrado a fim de facilitar a correção do teste. Cada matéria faz parte de uma única disciplina. A série está ligada a matéria.

Por exemplo, para a disciplina de matemática, Marina prepara um teste com 20 questões, cada questão corresponde a um bimestre (1, 2, 3 ou 4) e a uma matéria (ex: adição, divisão, números primos...). Cada matéria corresponde a um disciplina (adição - matemática, sinônimos - português).

Faça o diagrama de classes.

Bibliografia

BLAHA, Michael; RUMBAUGH, James. Modelagem e projetos baseados em objetos com UML 2 - 2ª edição. São Paulo: Elsevier, 2006.

OMG. About the Unified Modeling Language Specification Version 2.5.1. Disponível em <https://www.omg.org/spec/UML/>. Acesso em: 07 de novembro de 2018.

Anexo - Respostas sugeridas das atividades práticas:

Atividade 1 - Resposta sugerida