PERCURSO DE APRENDIZAGEM 1

Universidade de Fortaleza | EAD

PROGRAMAÇÃO ORIENTADA A OBJETOS

REPRESENTAÇÃO DO MUNDO REAL SOB A PERSPECTIVA DA ORIENTAÇÃO A OBJETOS

Universidade de Fortaleza | EAD

REPRESENTAÇÃO DO MUNDO REAL SOB A PERSPECTIVA DA ORIENTAÇÃO A OBJETOS

Sumário

Sumário clicável

Você já parou para pensar que quase tudo que fazemos no nosso dia a dia exige alguma interação com sistemas? No caminho para a universidade você pode utilizar o waze para ver as melhores rotas ou utilizar aplicativos de transporte como 99pop ou uber, sem contar com os sinais de trânsito e com os fotossensores que são controlados, também, por softwares. Ou seja, é inevitável hoje a nossa interação com os sistemas computacionais. É com base na reflexão para essa pergunta que damos o ponto de partida para este material. Como podemos utilizar a Programação Orientada a Objetos para simular eventos do dia a dia em sistemas digitais. Para tanto, nesta unidade, aprenderemos o conceito e a importância do estudo da orientação a objetos e de seus pilares. Faremos uma discussão sobre paradigmas de linguagens de programação e linguagens de programação, mais precisamente abordando o paradigma Orientado a Objetos (POO) e Java, e sua representação através de UML e seus diagramas. Por fim, escreveremos nossos primeiros programas no paradigma OO para, em seguida, dar início à manipulação de atributos e métodos e na representação das associações dentro das classes.

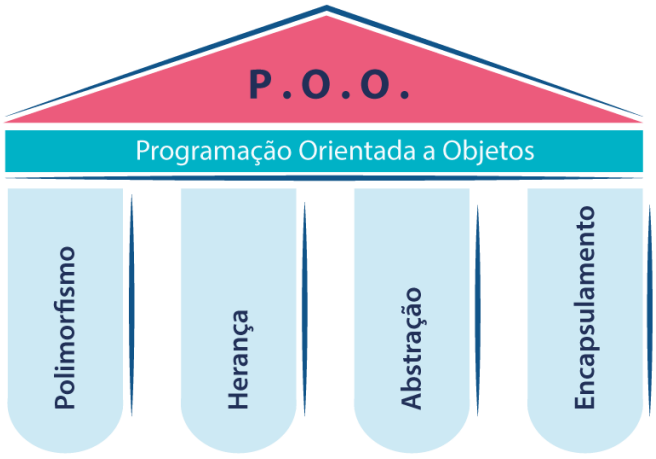
Olá

1. Pilares da orientação a objetos

Damos início a esta unidade diretamente com uma curta pergunta: o que é um **paradigma de linguagem de programação orientado a objetos**? Você já viu essa expressão na *internet* ou em alguma rede social? Define-se paradigma de linguagem de programação, no contexto de desenvolvimento de *software*, como sendo um modelo para estruturar e representar problemas, cuja solução deseja-se obter por meio de um programa, construído a partir de uma linguagem de programação (SEBESTA, 2018). Os principais paradigmas são: imperativo, lógico, funcional, orientado a eventos e orientado a objetos, entre outros. Em algumas linguagens de programação, dependendo dos recursos oferecidos, é possível programar em mais de um paradigma. Neste capítulo, estudaremos o principal paradigma de programação utilizado no momento da escrita deste texto: o Paradigma Orientado a Objetos (POO) e concretizaremos os exemplos em Java, uma das linguagens de programação mais utilizadas para desenvolvimento de *software* (TIOBE, 2021).

Para entendermos do que se trata a orientação a objetos, é essencial entender em quais conceitos ele se baseia (vide figura 1). Para tanto, a linguagem precisa atender a quatro características muito importantes: Polimorfismo, Herança, Abstração e a última, e não menos importante, Encapsulamento. A figura 1 ilustra os quatro principais pilares do POO, os quais serão descritos nas próximas seções.

Figura 1 - Pilares da orientação a objetos



Fonte: Florenzano, 2021.

1.1 Abstração

Assim como acontece em vários cenários da vida, abstraímo-nos de certas dificuldades para atingirmos nossas metas. Não poderia ser diferente na programação orientada a objetos. Afinal, como foi visto na disciplina de algoritmos: programamos para automatizar processo do nosso dia a dia.

A ideia por trás dessa característica está em que não devemos levar em conta propriedades e características importantes para o domínio do problema e se abstrair daquelas que não são relevantes para o problema.

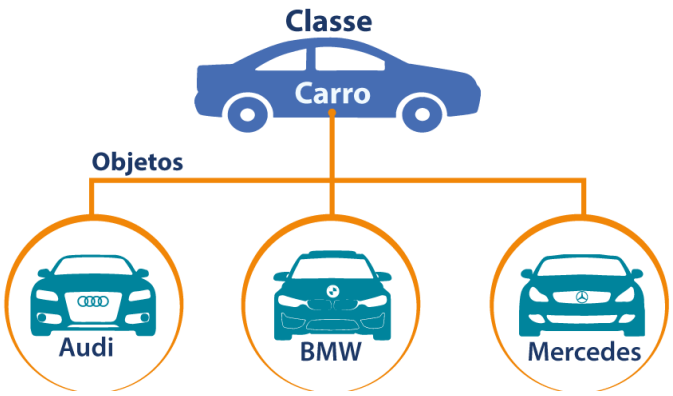
Para entender melhor essa situação, vamos pensar no seguinte problema: Fomos contratados para implementar um sistema para gerenciar uma concessionária de carros. Dessa forma, evidentemente, o sistema deverá representar a principal entidade desse contexto, no caso, o objeto carro e todas as características dele. No caso, esses objetos (instâncias das classes) deverão ter várias propriedades importantes, tais como: cor, modelo, quantidade de portas, marca, ano de fabricação, ano do modelo, dentre outras, uma vez que o produto principal dessa empresa é a venda de carros e todas as características do carro são essenciais. Para uma empresa pública, como o DETRAN ou DENATRAN, também seria importante todas essas características do "objeto" carro.

Veremos nas próximas seções que um modelo de uma entidade (chamada de objeto até por razões óbvias do nome do paradigma) será denominada de classe e as instâncias desse modelo serão os objetos (vide figura 2), que demonstram o modelo do carro e versões concretas desse modelo. No caso, temos um carro e tipos de carros Mercedes, Audi e BMW.

Agora, pensando, em outro contexto de problema, o domínio de um sistema para gerenciamento da vacinação da COVID-19. Nesse caso, se a única informação relevante para o sistema fosse se o usuário tem ou não carro para que possa optar pelo *drive-thru* da vacinação ou não. Contudo, não é relevante para esse sistema armazenar informações específicas do carro, como número do chassi, ano de fabricação, como nos dois outros sistemas citados.

É importante notar que a mesma entidade seria representada de forma diferente nos três sistemas, com mais ou menos características, apesar de representarem o mesmo objeto. Esse aspecto é uma grande vantagem, pois nos permite, a partir de um contexto inicial, modelar necessidades específicas. Isso possibilita flexibilidade no processo de programação, pois é possível não trabalharmos com o todas as propriedades da entidade modelada, mas sim com suas abstrações.

Figura 2 - Representação da entidade carro



Fonte: Progressiva, 2021.

Importante!

Vale ressaltar que alguns autores só consideram como pilares da POO: encapsulamento, herança e polimorfismo. Porém, outras referências consideram a abstração como um dos principais pilares.

1.2 Encapsulamento

Uma das principais propriedades da Orientação a Objetos, senão a principal, é o encapsulamento. A ideia desse pilar está representada no próprio nome: encapsular no sentido de o que se deve "colocar dentro da cápsula", no caso, da entidade representada pelo modelo, ou seja, que propriedades e quais ações são essenciais para a representação do problema. Nos próximos circuitos, veremos que essas propriedades serão chamadas de atributos e as ações de métodos. A tabela 1 exemplifica três entidades com suas respectivas características e ações. Lembrando, como visto na seção anterior, as colunas 2 e 3 dependem do contexto do problema, ou seja, do primeiro pilar: a abstração.

Tabela 1 - Abstrações do mundo real

Entidade	Características	Ações
Conta bancária	número, saldo, agência	Depositar, sacar, extrato
Carro	cor, marca, portas, peso	ligar, desligar, acelerar
Celular	marca, memória, tamanho	ligar, desligar, volume

Fonte: Elaborada pelo autor.

Uma analogia seria: se você fosse a um atendimento médico e lhe fosse receitado:

- 250 mg de *Saccharomyces boulardii* CNCM I-745 ;
- Liofilizado (250 mg de liofilizado contém no mínimo 1,25 x 109 células de *Saccharomyces boulardii* CNCM I-745).
- Excipientes: estearato de magnésio e lactose.

Seria muito complexo para uma pessoa que não tivesse formação em farmácia ou química usar essa informação para manipular o remédio. Por isso, o mais simples seria abstrair essa composição e simplesmente receitar o remédio 'X', ou seja, o encapsulamento esconderia a complexidade. As cápsulas de um remédio seriam um exemplo de como encapsular ingredientes para um fim específico, sem necessariamente se preocupar com compostos e os métodos de manipulação dos mesmos dentro das cápsulas (vide figura 3).

Uma vantagem deste princípio é que as mudanças se tornam transparentes, ou seja, quem usa algum processamento não será afetado quando seu comportamento interno mudar. Linguagens estruturadas preveem este fundamento, mas para atingi-lo é mais difícil. Os conceitos de classe, método, entre outros, facilitam bastante a aplicação deste fundamento.

Esse é o principal conceito de OO, uma vez que as modificações ficam transparentes para os usuários do código, as funcionalidades ficam encapsuladas dentro de um local, onde os usuários desses serviços podem se abstrair. E alterações que por ventura tiverem que ocorrer podem ficar concentradas em apenas um lugar. Um problema enorme para desenvolvimento de código é quando ocorre uma mudança de requisito, e isso implica em várias alterações no código pelo mesmo motivo, ou seja, o conceito de encapsulamento e abstração não foram bem utilizados.

O encapsulamento é o principal pilar da OO e, ao mesmo tempo, o mais complexo de ser utilizado de forma correta, uma vez que é necessária bastante experiência do programador para conseguir encaixar de forma correta as ações e propriedades nas entidades corretas.

Por exemplo, no sistema de gerenciamento de turmas em uma universidade, onde deveria estar localizada a ação de trancar disciplina ou trancar o curso, ou mesmo a ação de alocar um professor na disciplina, ela deveria estar em qual parte do sistema? Essas são questões que só com experiência e vivência um analista ou desenvolvedor saberia responder de forma segura, além de não ter uma resposta única.

Portanto, apesar do encapsulamento ser, dentre os pilares, uns dos mais simples de explicar ou de se entender, na prática, é um dos mais difíceis de colocar em prática de forma correta.

Figura 3 - Encapsulamento dos ingredientes

Fonte: Febrafar, 2017.

1.3 Herança

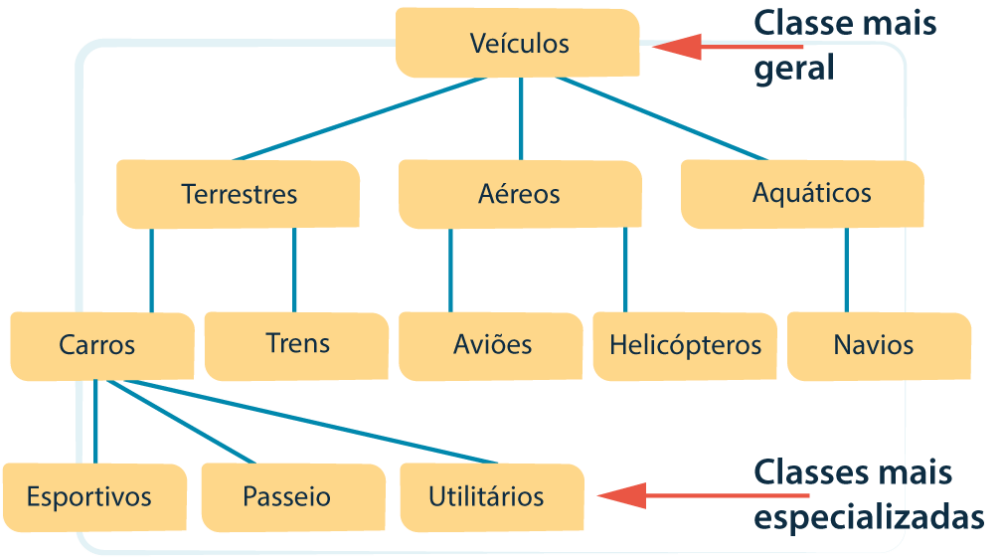
O encapsulamento não é exclusivo da orientação a objetos, uma vez que a programação por tipos abstratos de dados segue esse mesmo conceito. O que torna a orientação a objetos única é o conceito de herança (MANZANO, 2014).

Na Programação Orientada a Objetos, o conceito de herança tem significado semelhante ao do mundo real. Assim, como um filho pode herdar algumas características do pai e, no caso da classificação, dos seres vivos, em que a Espécie herda as características do Gênero e, por sua vez, o gênero herda as propriedades da família.

No contexto da OO, a herança ocorre entre classes, visto que as propriedades e as ações são herdadas entre elas.

Herança é um mecanismo que possibilita propriedades e ações comuns serem organizadas em uma classe base, com as características mais genéricas, ou superclasse ou classe pai. A partir de uma classe base, outras classes podem ser especificadas, conforme podemos ver na figura 4, na qual a classe Veículos é a classe base e os Esportivos, Passeios e Utilitários são classes mais especializadas. Cada classe derivada ou subclasse herda as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela (MANZANO, 2014).

Figura 4 - Exemplo de hierarquia de classes



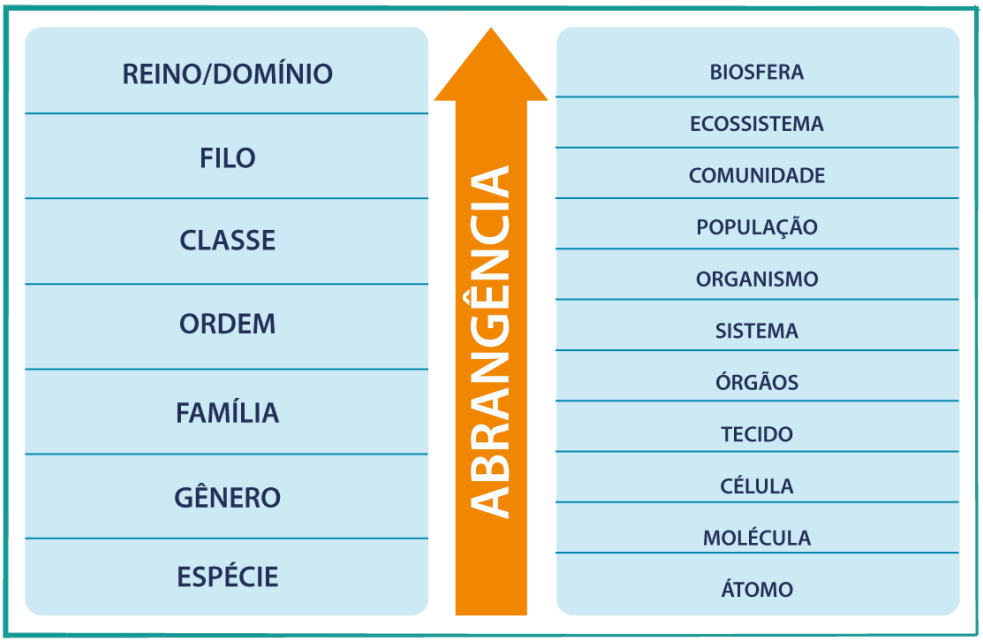
Fonte: Perrotti, 200?.

A herança pode ocorrer em quantos níveis forem necessários. Porém, uma boa quantidade de níveis é de, no máximo, 4. Quanto mais níveis existirem, mais complexo de entender o código ficará, pois cada vez mais um distanciamento do conceito base é gerado. Esses níveis são chamados de Hierarquia de Classe.

No exemplo da “classificação biológica” (vide figura 5), partindo do nível Espécie até Reino, a Hierarquia de Classe teria 6 níveis. E partindo do átomo até a biosfera, teríamos 12 níveis.

Por fim, a herança auxilia no reúso de código, uma característica muito importante no desenvolvimento de código, uma vez que as características e ações implementadas na classe base são “levadas” para as classes descendentes e, caso haja uma modificação na classe pai (base), ele é estendida para as classes filhas ou derivadas.

Figura 5 - Exemplo de hierarquia de classes



Fonte: CESCHIM, 2020.

1.4 Polimorfismo

Em determinados momentos em uma hierarquia de classes, precisamos que uma mesma ação (nome e lista de parâmetro, ou seja, assinatura) se comporte de forma diferente, dependendo de quem vai praticar essa ação na hierarquia. Isto é necessário devido às particularidades que as classes especializadas precisam exercer. Por exemplo, na Figura 2, que ilustra os carros, pode ocorrer que a ação de frear seja diferente, dependendo do carro, um use um sistema de freios mais moderno (sistema de freio ABS) e outro tipo de carro mais popular use o sistema a tambor, ou seja, a mesma ação de frear irá acionar mecanismos diferentes, apesar do resultado final esperado ser o mesmo, que é parar o carro (MANZANO, 2014).

Esta possibilidade de uma mesma ação poder se adaptar de acordo com a entidade que a executa é uma forma de polimorfismo. Outra forma de polimorfismo seria a capacidade de um objeto (instância da entidade) poder ser referenciado de várias formas.

EXEMPLO:

No caso da Figura 4, um objeto “Carros” ou “Navios” poderia ser referenciado por Veículos; já objetos da entidade “Esportivos, Passeio e Utilitários” poderiam ser referenciados por Carros, Terrestres ou Veículos. Essa característica é muito importante para abstração, uma vez que qualquer objeto ilustrado na Figura 4 poderia ser referenciado pela classe Veículo.

A grande vantagem do uso do polimorfismo é a flexibilidade, uma vez que podemos utilizar objetos distintos para executar a mesma ação, sendo que esta ação se moldará ao objeto que a executa.

Cuidado, polimorfismo não quer dizer que o objeto ficará mudando, um objeto é criado de um tipo, morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele (CAELUM, 2019).

Importante!

Tenho duas classes completamente diferentes, que não fazem parte de uma mesma hierarquia de classe. Contudo, elas possuem um método com a mesma assinatura (nome e parâmetros), de forma semelhante, mas com comportamentos diferentes. Isto é apenas uma grande coincidência, definitivamente, não é polimorfismo.

Outro ponto importante é que o polimorfismo traz a flexibilidade supracitada, porém, caso esta não seja necessária, a sua utilização pode ser prejudicial. Apenas com o amadurecimento dos pilares da herança e do polimorfismo, o desenvolvedor poderá utilizá-los de forma efetiva, pois, conforme veremos nos próximos circuitos de aprendizagem, a utilização da herança deverá ser preterida, em várias situações, por conta da utilização de composição. Além de que a utilização de herança em uma modelagem não obriga a utilização de polimorfismo, ou seja, não existe uma “fórmula mágica” para utilização desses pilares. A utilização correta das características da POO só ocorrerá com o amadurecimento e a experiência do programador ou analista em vários projetos que se utilizem dessas técnicas.

Dois pilares que se confundem muito é a abstração e o encapsulamento, porém eles são bem distintos. O primeiro está relacionado com o nível de modelagem do sistema e a sua contextualização dentro do contexto de sua aplicação, enquanto que o segundo “esconde” o funcionamento interno relacionado à implementação e ao código.

Por fim, é importante salientar que a POO é mais do que um paradigma de programação, uma vez que ela é utilizada, também, na modelagem de sistemas, utilizando a linguagem UML (FOWLER, 2011), baseado no pilar da abstração, conforme poderemos ver no Circuito de Aprendizagem 3.

2. Comparação entre os paradigmas estruturados e orientados a objetos

Existem diversos tipos de paradigmas, os quais podem ser subdivididos em imperativo e declarativo de forma didática. O paradigma imperativo é o mais comum e mais utilizado, uma vez que é aquele no qual o foco na resolução do problema é "instruir" o computador a executar as instruções passo a passo, como o próprio nome diz, de forma imperativa (mandatória). Pode-se ver no exemplo abaixo um algoritmo em forma imperativa, sendo este responsável por ditar os passos para sacar um valor do caixa eletrônico.

EXEMPLO:

- Passo 1: insira o cartão no caixa eletrônico;
- Passo 2: selecione a opção de saque;
- Passo 3: digite o valor a ser sacado;
- Passo 4: caso não possua o valor desejado em conta, retorne ao passo 3. Do contrário, siga para o passo 5;
- Passo 5: digite a senha de segurança;
- Passo 6: remova as cédulas do caixa eletrônico;
- Passo 7: retire o cartão.

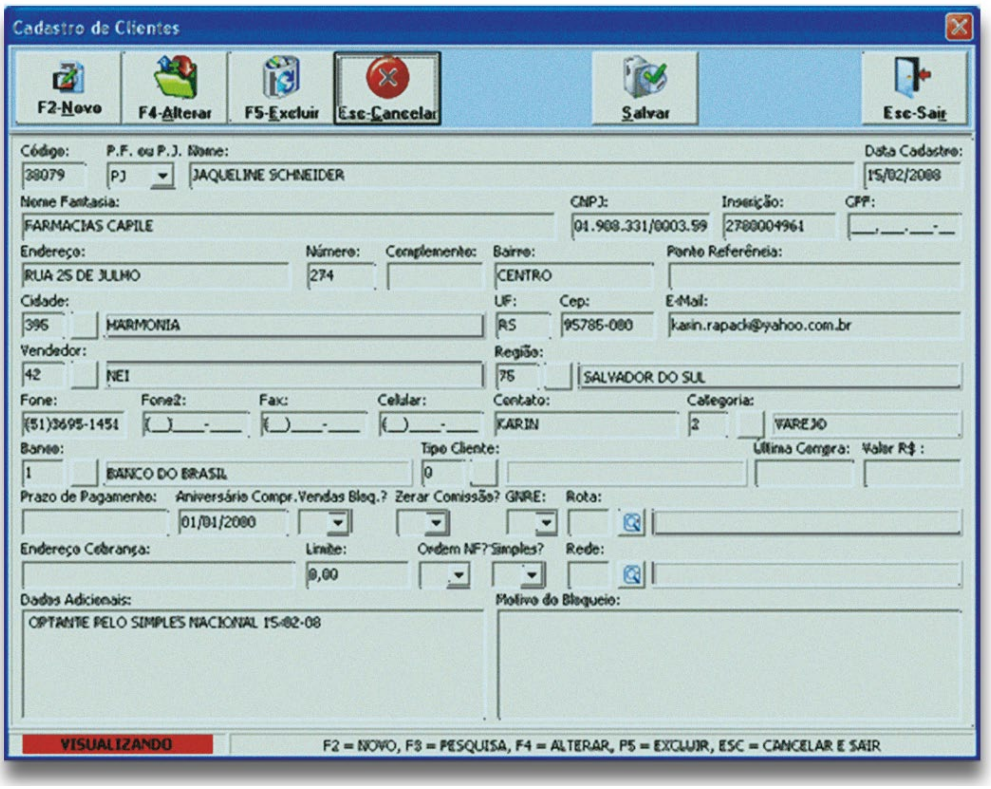
Dos paradigmas da programação imperativa, um se tornou o mais usado para iniciantes da programação: o Paradigma Estruturado. Este foi adotado pela sua simplicidade, como se pode perceber no exemplo do caixa eletrônico. Ele quase possui uma transformação direta entre o algoritmo e a programação nas linguagens desses paradigmas. As linguagens mais usadas que seguem esses paradigmas são C e Pascal. Em relação ao paradigma declarativo, esse padrão tem um desempenho computacional bem mais eficiente. Devido a isto, foram as linguagens mais utilizadas nos primórdios da computação, uma vez que o poder computacional era bem menos robusto.

O paradigma estruturado, também conhecido como procedural, foi o mais utilizado na década de 90 (SEBESTA, 2018). Porém, com o passar do tempo, as necessidades dos sistemas e o poder computacional, tanto dos computadores como dos aparelhos eletrônicos, foram aumentando rapidamente e, com isso, a necessidade foi alterada absurdamente. Ou seja, a demanda por formas de se expressar as novas necessidades

cresceu, além de o paradigma estruturado apresentar vários problemas que o paradigma OO se propõe a resolver, uma vez que o POO tem como principal característica a possibilidade de se criar códigos mais próximos das linguagens de alto nível (linguagens mais próximas das linguagens naturais), assim facilitando o desenvolvimento de sistemas de grande porte, principalmente, no ponto de vista da manutenção.

Para entender os problemas do paradigma procedural, é necessário contextualizar que na década de 90 as equipes de desenvolvimento eram pequenas e trabalhavam com formulários longos, vide Figura 6, como exemplo. O mesmo programador era responsável pela construção da tela, das validações, das regras de negócio e da persistência no banco de dados. Ou seja, anteriormente, trabalhávamos com um "*Euquipe*". Portanto, não havia como atentar para todos os detalhes. Porém, atualmente, temos programas muito complexos com equipes de 10, 20 pessoas, e o sistema de formulário se tornou insustentável.

Figura 6 - Tela de um formulário em delphi



Fonte: Active Delphi, 2016.

Orientação a objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural (CAELUM, 2005).

Para entender os problemas da programação procedural, vamos considerar o velho problema da validação do dígito verificador, que poderia ser de um CNPJ ou um CPF. Normalmente, temos um formulário para receber essa informação e uma função para validar esse dígito verificador, conforme a tabela 2.

Tabela 2 - Exemplo de validação de código

cpf = formulario-.campo_cpf valida(cpf)	cnpj = formulario.campo_cnpj valida(cnpj)

Fonte: Elaborada pelo autor.

O problema é que em todos os formulários que fôssemos receber o CPF ou o CNPJ para cadastro, por exemplo, ou para consulta, seria necessário validá-los, e isso geraria redundância de código relativamente grande. Além de termos que gerenciar o problema de toda tela em que fosse necessária a inserção ou a consulta de um CPF ou CNPJ teria que ter essa validação.

Poderíamos, por exemplo, criar um manual do sistema da empresa para os novos funcionários, mas essa não é a alternativa mais simples. O ideal é que possamos fazer uma alteração em um único local do sistema e, assim, os CPFs e CNPJs em todas as interfaces de usuário precisariam ser validados.

Para facilitar o entendimento, veja a Figura 7 (a), (b) e (c), onde teríamos um exemplo no qual o CPF deveria ser validado em três locais diferentes. Ou seja, se em algum desses locais o programador não lembrasse de chamar a validação, teríamos um bug no sistema. E o pior, se por algum motivo houvesse uma alteração no algoritmo de validação do CPF, por exemplo, a quantidade de dígitos fosse aumentada ou o algoritmo de criação do dígito verificador fosse modificado, teríamos que localizar todos os lugares onde essa validação é executada e alterar. Se, por ventura, em algum desses lugares, fosse esquecido de fazer essa alteração, teríamos outro problema no sistema difícil de detectar, pois usuários poderiam incluir CPFs ou CNPJs inválidos.

Um outro ponto que poderia ser adicionado a esse problema seria uma regra da LGPD (Lei Geral de Proteção de Dados), que não permite que, para CPFs diferentes, seja adicionado e-mails iguais. Ou seja, teríamos um código conforme a Tabela 3.

Tabela 3 - Exemplo de validação de email

cpf = formulario.campo_cpf
email = formulario.campo_email
valida(cpf)
....
se e-mail não existente
  insere(cpf, e-mail, ...)
senão alerta_erro(email_existente)
	cnpj = formulario.campo_cnpj
	e-mail = formulario.campo_email
	valida(cnpj)
	....
	se e-mail não existente
	  insere(cnpj, e-mail, ...)
	senão alerta_erro(email_existente)

Fonte: Elaborada pelo autor.

Toda vez que fôssemos inserir um e-mail vinculado a um CPF ou CNPJ, teríamos que lembrar de fazer a validação da Tabela 3. É provável que você esqueça de chamar esse validador. Principalmente se o seu sistema tem 50 formulários e precise validar em todos os CPFs e CNPJs esses e-mails e, além disso, sua equipe tem quatro programadores trabalhando nessas funcionalidades. Teoricamente, todos têm que ficar responsáveis por essa validação!

A situação pode piorar se um novo desenvolvedor entrar na equipe, pois precisaría-se avisá-lo de que sempre precisamos validar um CNPJ ou CPF e os e-mails vinculados a eles. E é nesse momento que começa a nascer aqueles wikis ou guias de programação para serem lidos pelos novos programadores do projeto, gerando uma documentação enorme e com uma probabilidade de ficar defasada enorme. Vale aquela premissa, melhor um código sem documentação do que um código com documentação desatualizada.

Resumindo, qualquer estagiário ou novo contratado teria que ler uma quantidade enorme de material para conseguir começar a desenvolver no projeto, desperdiçando tempo precioso que poderia ser alocado para entendimento de regras de negócios específicas do sistema ou para entendimento da tecnologia ou padrões de projetos utilizados no projeto.

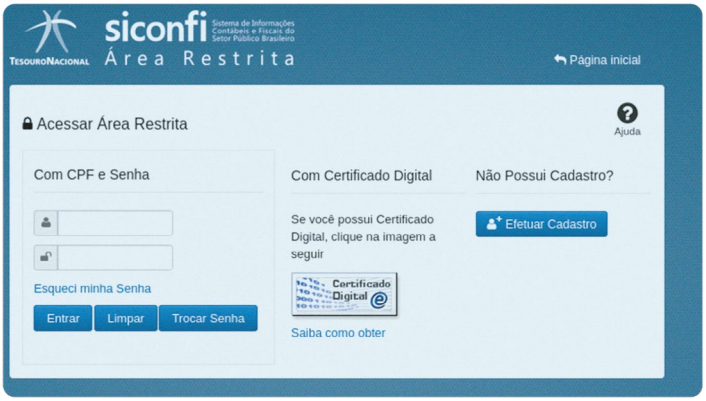
Com a orientação a objetos, a ideia de dados e as funcionalidades - ou “comportamentos” - estarão interligadas, gerando uma enorme facilidade na organização e na manutenção de um determinado sistema. Ou seja, para a validação do dígito verificador do CPF ou do CNPJ ou a validação para inserção ou alteração do e-mail estariam **encapsulados** (aqui começamos a enxergar a mágica da POO, através do seu pilar mais importante) na classe Usuário ou na classe empresa. Ou seja, toda vez que fosse necessária a digitação do CPF ou CNPJ, ele sempre seria validado, e essa validação estaria apenas em um lugar em todo o sistema, ocorrendo a mesma ideia para a alteração ou inserção de um e-mail vinculado a uma empresa ou usuário.

Na programação procedural, é muito comum ocorrer uma situação em que um desenvolvedor que irá fazer manutenção de um código de um terceiro precise ler o código para verificar os detalhes e descobrir como ele funciona internamente. Um sistema bem encapsulado não deveria gerar essa demanda. É impraticável para um sistema real ter que entrar nos detalhes da implementação, pois estaríamos indo de encontro ao paradigma da abstração.

Adicionalmente, imagine que, em todos os formulários da figura 7 (a),(b),(c), você também tivesse que validar a idade do usuário – exemplo o usuário precisa ter mais de 21 anos. Seria necessário colocar um se (if)... Mas onde? Espalhado por todo sistema e, mesmo que se crie outra função (método) para validar, teríamos que repetir isso em todos os formulários. Qual é a chance de faltar a atualização em um deles? Enorme.

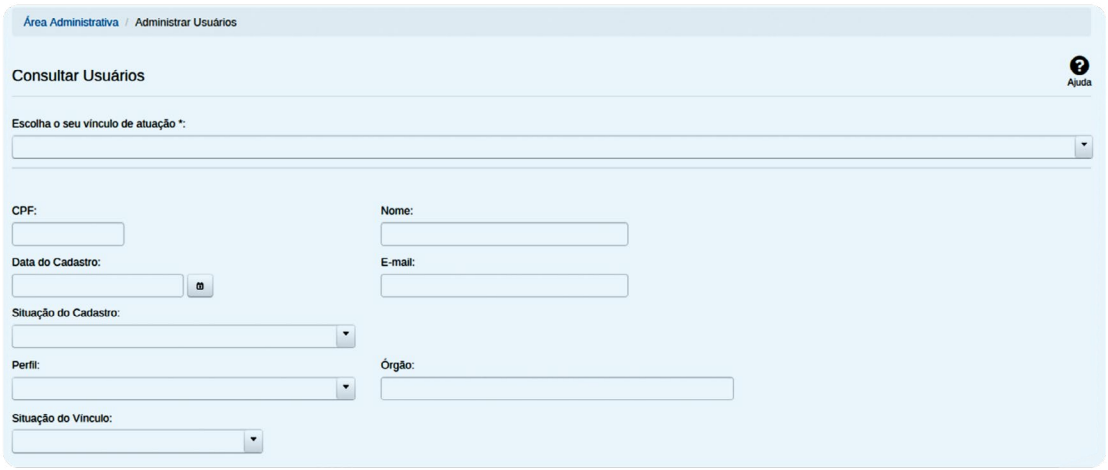
A responsabilidade de verificar se o cliente tem mais de 21 anos ficou espalhada por todo o seu código. A ideia aqui seria concentrar isso em um local único, com o intuito de viabilizar o reúso do código.

Figura 7 (a) - Tela de login do sistema *siconfi*



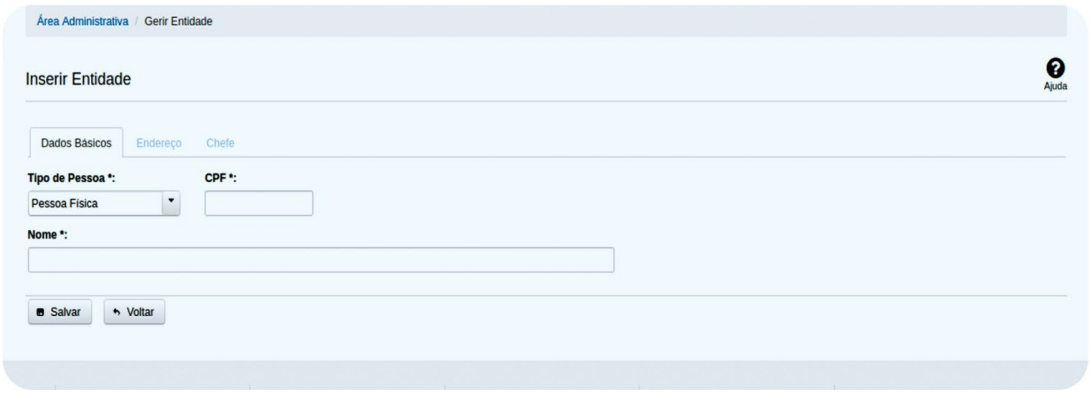
Fonte: SICONFI, 2019.

Figura 7 (b) - Tela de login do sistema *siconfi*



Fonte: SICONFI, 2019.

Figura 7 - Tela de login do sistema *siconfi*



Fonte: SICONFI, 2019.

O ideal seria se conseguíssemos alterar essa validação e ficasse transparente para outros desenvolvedores. Em outras palavras, eles construiriam formulários utilizando CPF, CNPJ e idade, e uma única função seria responsável por isso: os outros programadores, teoricamente, nem saberiam da existência desse trecho de código. Impossível? Não, o paradigma da orientação a objetos viabiliza tudo isso.

Um dos problemas do paradigma estruturado é que não temos uma forma simples de criar conexão forte entre dados e funcionalidades. No paradigma orientado a objetos, é muito simples esse relacionamento por meio dos recursos da própria linguagem. A programação orientada a objetos viabiliza tudo isso.

Existe uma diferença muito importante entre o reaproveitamento de informações entre as linguagens estruturadas e a orientação a objetos, já que na primeira o encapsulamento ocorre entre dados e, no segundo modelo, ocorre em relação a dados e funcionalidades, como vimos no circuito de aprendizagem anterior. No caso do paradigma procedural, temos apenas a possibilidade de variáveis locais e globais, ou seja, ficamos muito limitados, uma vez que as variáveis locais têm um escopo muito limitado (somente dentro daquela função especificamente) e as variáveis globais não viabilizam o reúso, além de serem definidas separadamente (desassociadas) das funcionalidades (ações).

Para esclarecer melhor, vamos para um exemplo em C, linguagem estruturada, e outro em Java, linguagem Orientada a Objetos.

Importante!

Vale ressaltar que nas versões atuais Java é uma linguagem multiparadigma, porém nela ainda predomina o paradigma OO, apesar de ser possível utilizar o paradigma funcional, por exemplo, notação lambda onde vamos encapsular uma entidade Conta.

Tabela 4 - Exemplo da implementação da entidade "Conta" na linguagem C (a) e na linguagem Java (b)

```
typedef stru ct{
//p rop riedades;
int nume ro;
int agencia;
Clie nte clie nte;
} C o nta;
....
// a ções (desassociadas dos dados)
creditar(floar valo r, C o nta c1);
debitar(floar valo r, C o nta c2);
trans fe rir(floar valo r, C o nta c1,
C o nta c2);
(a)
```

```
class C o nta{
//p rop riedades;\
int nume ro;
int agencia;
Clie nte clie nte;
//a ções
creditar(floar valor);
debitar(floar valor);
trans fe rir(floar valo r, C o nta c);
}
(b))
```

Fonte: Elaborada pelo autor.

Como podemos ver na tabela 4 (a), o código em uma linguagem estruturada só agrega (encapsula) dados; as ações (funcionalidades) são disjuntas, separadas. Portanto, para creditar um valor em uma conta, é necessário passar a conta por parâmetro. Exemplo: creditar (100,00, conta1), ou seja, creditar 100 reais e especificar explicitamente a conta1.

Por sua vez, em uma linguagem OO, as ações e os atributos estão "juntos", associados, então podemos chamar conta1.creditar(100,00), implicitamente. Comparando os dois exemplos, fica claro que a opção 2 é mais coesa, uma vez que podemos perceber que a entidade conta tem como característica creditar valores nela. Na situação (a), poderíamos ter as funções creditar, debitar e transferir em um arquivo e a definição dos dados em outro, ou seja, as ações e funcionalidades ficariam em locais distintos e os desenvolvedores poderiam facilmente se perder. Na situação (b) isso não ocorre, uma vez que o programador teria que definir dentro da *class* Conta as propriedades e as ações inerentes a ela.

Outro ponto importante seria a análise do método transferir: na situação (a), é necessário ter em mente que a transferência ocorrerá entra a conta c1 e c2 (transferir(100, c1, c2), mas de qual conta a transferência ocorrerá da c1 para c2 ou o contrário. Na situação (b), fica bem mais legível (legibilidade é provavelmente a característica de linguagem de programação mais relevante em linguagens de programação, uma vez que na maior parte do tempo um sistema fica na fase de manutenção - já que a transferência ocorrerá a partir da conta que está executando a ação para a conta passada por parâmetro). Essa é a "mágica" da POO para encapsular dados e ações em um mesmo local.

Claro que a linguagem sozinha não resolve os problemas, pois se o codificador "forçar a barra" ou não souber o que está fazendo, ele pode programar estruturadamente em uma linguagem OO. Vide Tabela 5.

Tabela 5 - Exemplos de duas implementações da entidade "Pessoa" na Linguagem Java

```
class Aluno {
                    //propriedades;
                      String cpf;
                      String nome;
                      Endereco endereco;
                    //ações
  static matricular(Disciplina d1, Aluno
                    a1);
  static trancar(Disciplina d1, Aluno a1);
   static trancar(Curso c1, Aluno a1);
}
(a)
```

```
class Aluno {
  //propriedades;
    String cpf;
    String nome;
    Endereco endereco;
  //ações
   matricular(Disciplina d1);
   trancar(Disciplina d1);
   trancar(Curso c1);
}
(b)
```

Fonte: Elaborada pelo autor.

No exemplo (a), temos uma implementação da class aluno na linguagem Java, que apesar de estar utilizando uma linguagem OO, a forma como foi implementada, passando por parâmetro nos métodos matricular e trancar o aluno que irá efetuar a matrícula ou trancá-la, perde um dos pilares da orientação a objeto, o encapsulamento.

Usando o código do exemplo (a) para um aluno se matricular em uma disciplina, é necessário executarmos o seguinte código *Aluno.matricular(disciplina_1, aluno_1)*, ou seja, a função matricular está dissociada da disciplina e do aluno; já na situação (b), para executarmos a mesma ideia, bastaria executarmos *aluno_1. matricular(disciplina_1)*, de modo que o objeto aluno_1 teria a responsabilidade de se matricular. Aproveitando, podemos verificar a utilização de outro pilar nesse exemplo: temos dois métodos com o mesmo nome: trancar, porém o primeiro tranca uma disciplina e o segundo o curso. Esse é um exemplo de Polimorfismo, no qual dois métodos com o mesmo nome têm comportamentos diferentes, uma vez que depende do parâmetro passado.

É comum depararmos com códigos como o da tabela 7 (a), principalmente, quando existem programadores que já têm muita vivência nas linguagens estruturadas e começam a desenvolver em linguagens eminentemente OO. Esse curso tem o intuito de ensinar como podemos utilizar o POO da melhor forma possível.

3. Diagrama de classe

Antes de entrarmos em detalhes em relação ao diagrama de classe, faremos uma contextualização da linguagem de modelagem UML.

1.1 UML

Para entender o que é UML, vamos inicialmente pensar quais são as etapas para o desenvolvimento de um *software?* - Já parou para pensar sobre isso!

Em um sistema real, é necessário termos várias etapas, desde a elicitação de requisitos, em que o analista, juntamente como o usuário, faz os levantamentos das funcionalidades e processos do sistema, até a etapa de implantação e testes.

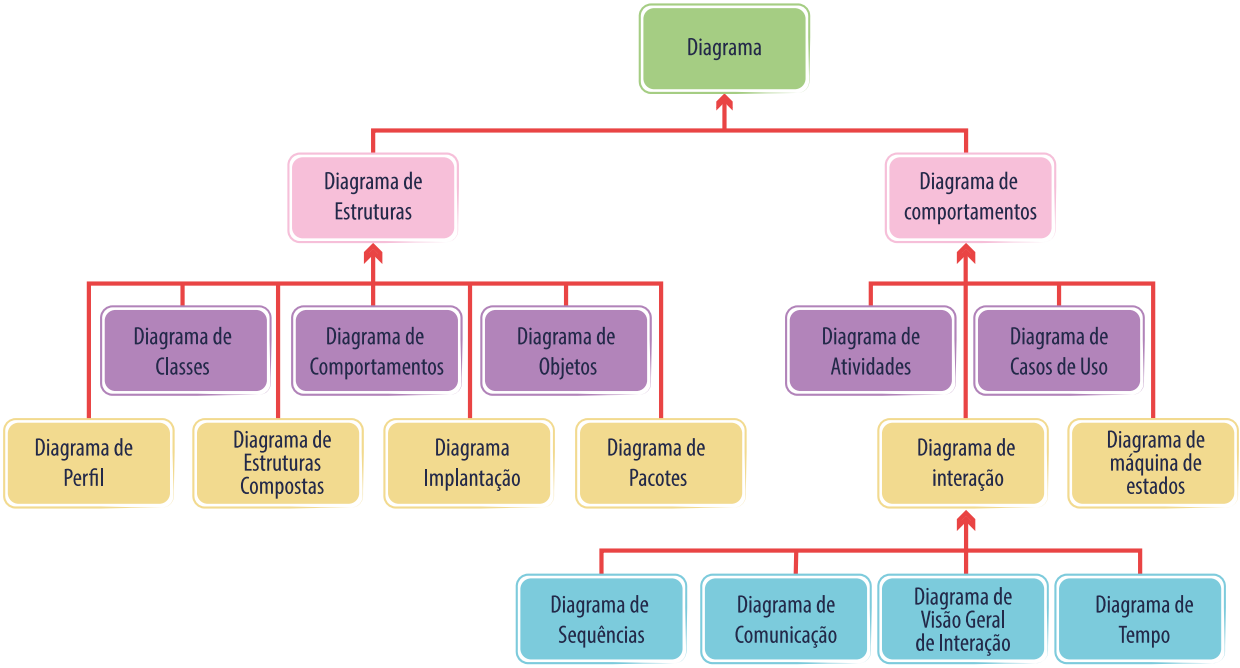
É nesse ponto que a UML é essencial, para modelar e documentar o sistema – *"UML? Mas o que é UML?"*

A *Unified Modeling Language*, ou Linguagem Unificada de Modelagem, é uma linguagem de modelagem, como o próprio nome indica, usada para ajudar na construção e na documentação das diversas fases do desenvolvimento de sistemas orientados a objetos. Ela surgiu da união de três métodos: o método de Booch, o método OMT (Object Modeling Technique), de Jacobson, e o método OOSE (Object-Oriented Software Engineering) de Rumbaugh.

Essas três metodologias foram as mais utilizadas na década de 90 e a UML veio como tentativa de padronizar a modelagem OO com apoio da *Rational Software*. Conforme a Figura 8, a UML 2.5 tem quinze diagramas divididos em:

- Diagramas de Estruturas ou Estáticos: descrevem os elementos estruturais que compõem o sistema, representando seus componentes e suas relações. Diagrama de pacotes, classes, objetos, estrutura, implantação, perfil e componentes fazem parte dessa classe.
- Diagramas de Comportamentos ou Dinâmicos: descrevem o comportamento dos elementos e suas interações e são divididos em: atividades, casos de uso, interação, máquina de estado, sequência, comunicação, tempo e integração.

Figura 8 – Diagrama UML

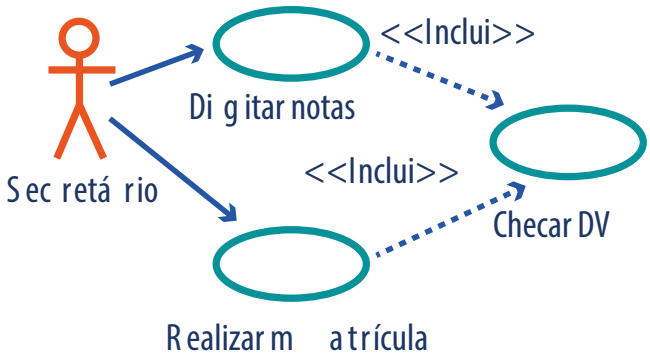


Fonte: Adaptado de *UML Superstructure Specification* 2.4.1, *Figure* A.5.

Entre os diagramas de comportamento, os mais importantes são os de caso de uso, sequência e atividades. O diagrama de caso de uso tenta, através de uma linguagem simples, demonstrar o comportamento de um sistema pela perspectiva do usuário. O próprio nome identifica a representatividade desse diagrama: "caso de uso" - exemplo de utilização do sistema.

Um caso de uso pode ser representado por uma elipse com sua respectiva ação. Atores são representados por "*stickman*" com seu papel específico, conforme podemos ver na figura 9, secretário.

Figura 9 – Exemplo de um diagrama de caso de uso para inclusão de notas



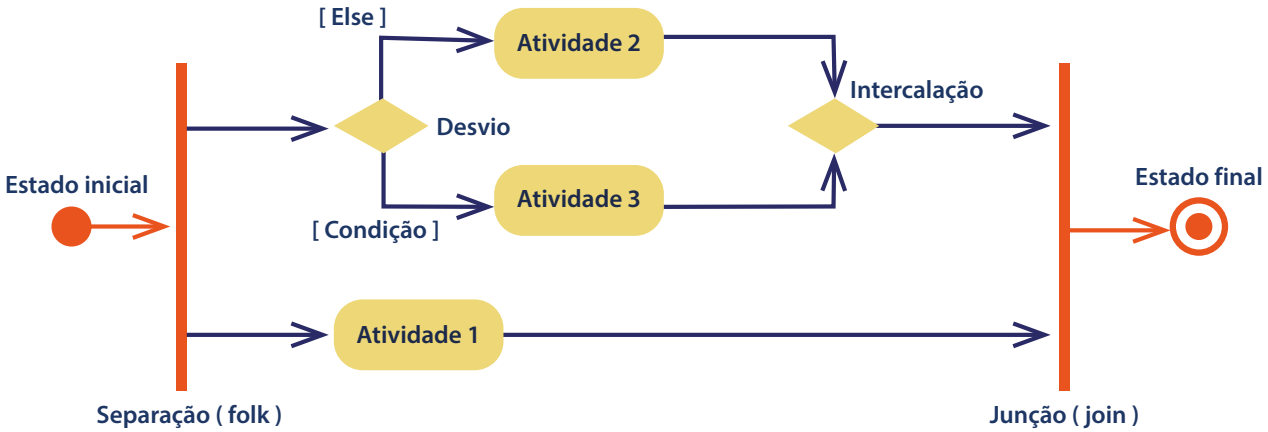
Fonte: Elaborada pelo autor.

Na figura 9, podemos ver o secretário realizar dois casos de uso: "digitar notas" e "realizar a matrícula", e o caso de uso de inclusão indica que a funcionalidade de checar o dígito verificador (DV) é compartilhado (reúso de código) entre os dois casos de uso. Podemos especificar o comportamento de um caso de uso através da descrição do fluxo de eventos, tais como:

1. Como e quando o caso de uso se inicia e termina;
2. Quando o caso de uso interage com os atores;
3. Fluxo básico (nele consideramos que não irá ocorrer nenhuma exceção, seria "*happy day*" – dia feliz, durante o qual todo o fluxo funcionaria;
4. Fluxos alternativos;
5. Fluxos de exceção.

Outro diagrama importante na modelagem de sistemas na UML é o diagrama de atividades que, a partir da versão 2.0 da UML, passou a ser diagrama independentemente. É o diagrama com maior ênfase no nível de algoritmos e, provavelmente, um dos mais detalhistas e tem semelhanças com os antigos fluxogramas, utilizados para desenvolver a lógica de programação. Na figura 10, temos um exemplo de diagrama de atividades, no qual podemos perceber o estado inicial e final, as raias de separação e junção, além dos pontos de decisão, ações (atividades) e transição.

Figura 10 – Exemplo do diagrama de atividades

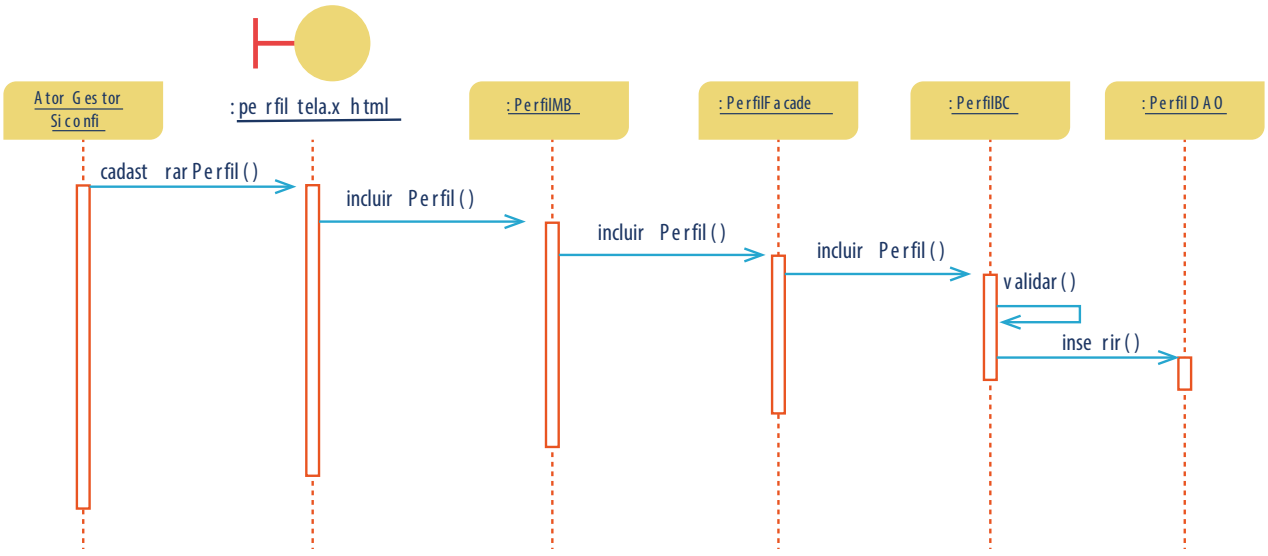


Fonte: Elaborada pelo autor, adaptada por EaD Unifor.

Temos, também, o diagrama de sequência, um diagrama dinâmico que dá ênfase à ordem temporal das mensagens, de modo que, onde a partir dele, percebe-se a sequência de mensagens enviadas entre os objetos. Para criarmos um diagrama desse modelo, seguimos os seguintes passos:

1. Definir os objetos da interação;
2. Definir a linha de vida do objeto;
3. Definir as mensagens trocadas entre os objetos.

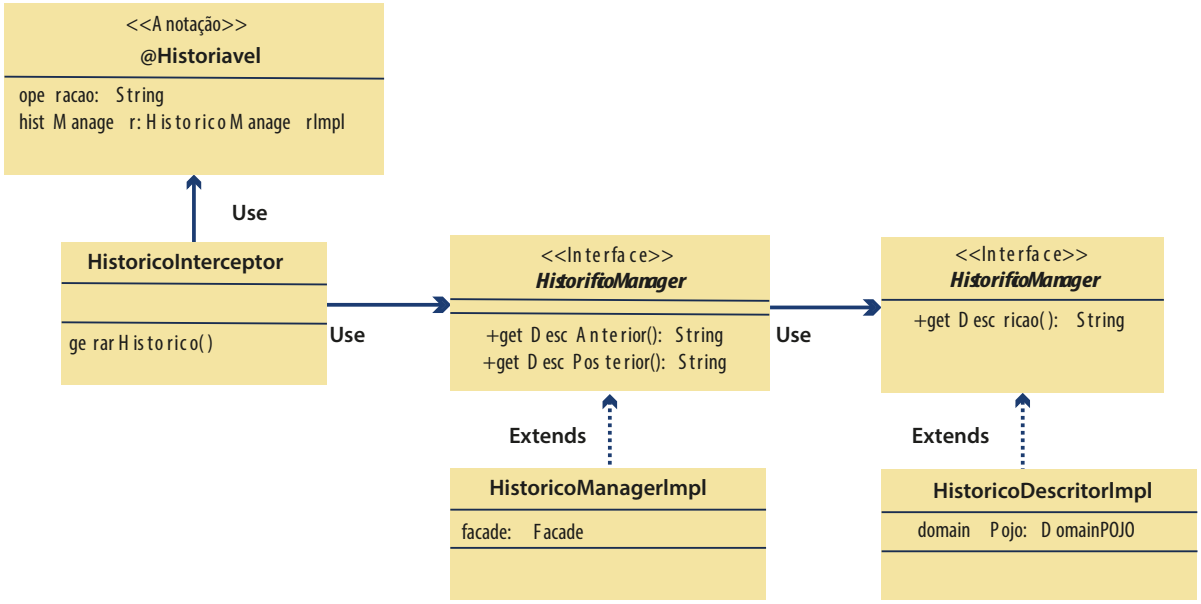
Figura 11 – Exemplo do diagrama de atividades para cadastrar perfil do sistema



Fonte: Elaborada pelo autor.

Na figura 11, podemos notar o diagrama de sequência evidenciando a chamada do método cadastrarPerfil/incluirPerfil passando por várias camadas do sistema. Em relação aos diagramas de estrutura, o mais importante e pertinente é o diagrama de classes, o qual é um dos modelos mais importantes na engenharia de *software* e serve de base para outros diagramas.

Figura 12 – Exemplo do diagrama de atividades para cadastrar perfil do sistema.



Fonte: Elaborada pelo autor.

Isso porque ele é utilizado para mapear as classes e seus relacionamentos dentro de um sistema, deixando claro o uso de herança, polimorfismo e, principalmente, encapsulamento. Na figura 12, podemos perceber a modelagem de várias classes: Historiavel, HistoricoInterceptor, HistoricoManager, dentre outras. Evidenciamos, também, a utilização de herança com o uso <<*extends*>> e de <<*interface*>> esses nomes entre <<_>> são denominados em UML como estereótipos, os quais adicionam semântica aos símbolos.

Para facilitar o entendimento do diagrama de classes, vamos explicar inicialmente os conceitos de classes e objetos em OO.

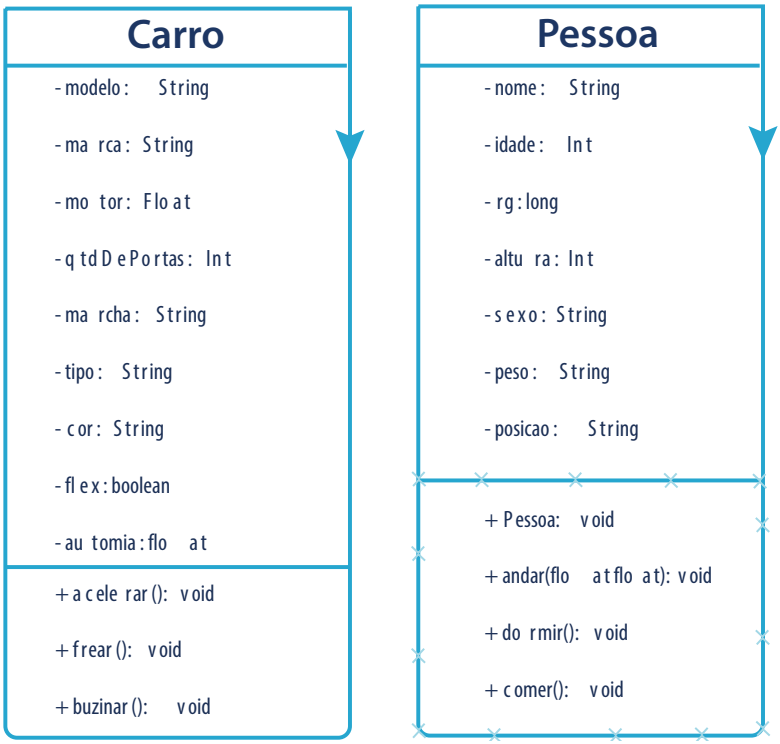
1.2 Classe e Objetos

A palavra classe vem da taxonomia em biologia. Os seres vivos de uma classe tem comportamento e características semelhantes, porém não são exatamente iguais, pois podem ter variações nesses dois fatores. A classe é um modelo, um molde de um objeto, uma analogia interessante seria a planta de uma casa. É uma casa? Definitivamente, não. Não podemos dormir dentro de uma planta de uma casa, mas apenas na instanciação dessa (a casa).

Pode parecer óbvio, mas a diferença entre classes e objetos é essencial para que o estudante de OO consiga entrar nesse mundo tão fascinante da modelagem e desenvolvimento de *software*. A UML define, por sua vez, tanto o diagrama de classe como de objetos, sendo que o primeiro é essencial para o estudo dessa componente curricular. A partir desse momento, entraremos nos detalhes desse assunto.

Na figura 13, temos a representação de duas classes, em um diagrama de classe, que representam dois modelos de objetos do mundo real: *Carro* e *Pessoa*. Ainda nesses pontos sem relacionamento, apenas no próximo Circuito de Aprendizagem entraremos nesse contexto.

Figura 13 – Diagrama de classes



Fonte: Elaborada pelo autor.

Para exemplificar alguns pontos, vamos codificar essas duas classes em Java, inicialmente a classe Pessoa:

```
1.    public class Pessoa {
2.    private String nome;
3.    private int idade;
4.    private long rg;
5.    private float altura;
6.    private String sexo;
7.    private float peso;
8.    private float[] posicao;
9.    public Pessoa() {
10.         posicao = new float[2];
11.   }
12. public void andar (float x, float y) {
13.       posicao[0] = x;
14.       posicao[1] = y;
15.  }
16. public void dormir() {
17.     System.out.println("Dormingo.");
18. }
19. public void comer() {
20.     System.out.println("Comendo");
21.  }
22. } // fechamento da classe
```

A linha 1 declara a classe Pessoa com suas propriedades (atributos) e métodos (ações). Na linha 9, temos a definição de um método especial denominado construtor, que será descrito em detalhes nos próximos percursos de aprendizagem. Na definição das propriedades, temos: idade, nome, rg, altura, sexo, peso e posição (vetor). Percebemos que no caso da linguagem Java, os atributos da classe, nesse momento de aprendizagem, podemos dizer que seriam "as variáveis" da classe, são todas tipadas (*float*, *int, String*, *float[]*) e temos, também, as ações implementadas: andar (linha 12), dormir (linhas 16) e comer (linha 19). Nesse contexto, o método andar supõe um deslocamento para o ponto x e y no eixo cartesiano. Isso vem de encontro à ideia da abstração e do contexto. Suponha-se que fosse a implementação de um sistema de jogos e que seria necessária a localização de uma pessoa no eixo cartesiano.

O Java é um linguagem tipada, diferentemente do JavaScript, portanto é necessário indicar os tipos dos atributos, variáveis e dos retornos dos métodos.

EXEMPLO:

```
int x = 10;
float f = 10.2f;
double d = 12.3;
char c = 'A';
```

No exemplo acima, temos a variável ***x*** do tipo inteiro, ***f*** do tipo float, ***d*** do tipo double e ***c*** do tipo caracter. No caso do JavaScript, o tipo das variáveis é inferido de forma implícita.

A seguir a implementação da classe Carro:

```
1.    public class Carro {
2.    private String modelo;
3.    private String marca;
4.    private float motor;
5.    private int qtdDePortas;
6.    private String tipo;
7.    private String cor;
8.    private float velocidade;
9.    public void acelerar(float aumento) {
10.         velocidade += aumento;
11.   }
12. public void frear() {
13.       velocidade = 0;
14.  }
15. public int getQtdDePortas() {
16.     return qtdDePortas;
17. }
19. public void buzinar() {
20.     System.out.println("Bi bi");
21.  }
22.   ...
23. } // fechamento da classe
```

A linha 1 declara a classe Carro com suas propriedades (atributos) e métodos. Nesse exemplo, podemos perceber que o atributo velocidade tem acesso restrito à classe (private), sendo que sua alteração está nos métodos frear, acelerar, que têm acesso public. Essa característica é muito utilizada em POO. As propriedades das classes são privadas e os métodos são públicos e têm a lógica de negócio definida neles. Ou seja, o frear já define que a velocidade do carro será 0, acelerar poderia ter uma restrição de velocidade máxima do carro ou da via, caso o atributo fosse público, isso seria mais complexo, uma vez que o tipo de dado float não limitaria a velocidade, podendo inclusive ter velocidade negativa.

Como podemos ver nos dois exemplos, o diagrama de classe tem alguns estereótipos para definir o encapsulamento dos atributos nas classes.

O "-" identifica a propriedade private, "+" o public, "#" o protected e "~" identifica o default. Por sua vez, os métodos têm o seu retorno definido após os dois pontos e os parâmetros entre parênteses: andar(float, float): float.

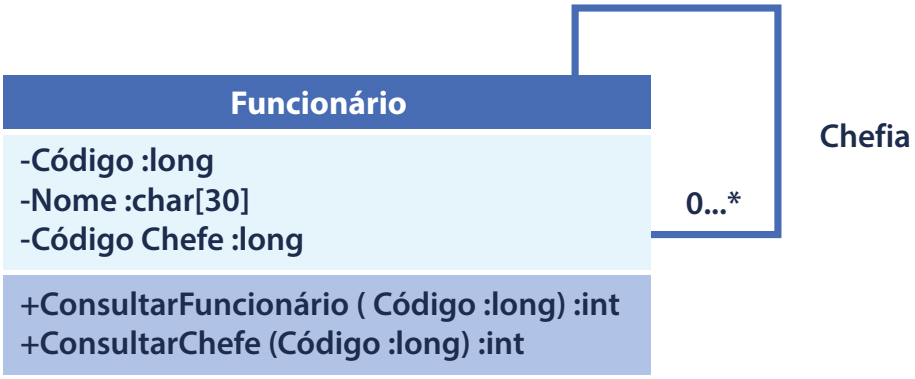
4. Relacionamento entre objetos: composição, agregação e associação

As classes costumam ter relacionamento entre si, com o objetivo de compartilhar informações e colaborar uma com as outras para poder mapear os processos do sistema. Existem diversas formas de relacionamento possíveis em um diagrama de classe: Associação, Agregação e Composição.

A associação descreve um vínculo que ocorre normalmente entre duas classes, especificando que os objetos de um item estão conectados a objetos de outro item. Nesse tipo de relacionamento, pode-se conectar uma classe a ela própria ou a outra classe. No autorrelacionamento, denominamos de associação unária ou reflexiva. A figura 14 mostra um exemplo desse tipo de relacionamento, em que um funcionário pode chefiar zero ou vários funcionários.

Vemos que os atributos código e código chefe refletem esse relacionamento entre o chefe e seus subordinados. O diagrama de classe e o modelo de entidade e relacionamento de banco de dados são bem semelhantes nesse sentido, e eles se complementam. Vale ressaltar que o atributo código de chefe pode ser omitido, uma vez que ele está implícito pelo relacionamento chefia.

Figura 14 – Exemplo de um relacionamento de associação unária ou reflexiva



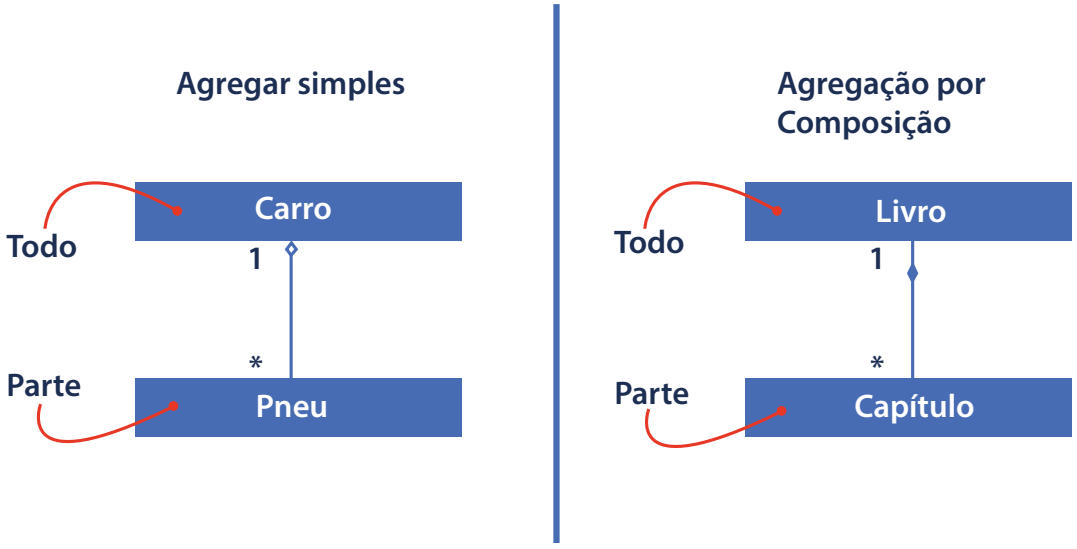
Fonte: GUEDES, 2006.

Nesse tipo de relacionamento, podemos definir outros elementos, como papéis, exemplo da "Chefia", a multiplicidade (0..*), o qual indica a quantidade de objetos que podem estar relacionados. Nesse caso, zero ou múltiplos (vários) funcionários podem ser chefiados por outro.

A agregação é um tipo especial de associação, na qual temos um relacionamento "todo/parte", em que o todo contém as partes. Esse tipo pode ser dividido em dois tipos especiais: a agregação simples ou por composição. A segunda é uma relação mais forte, na qual as partes não existem sem o todo.

Para facilitar o entendimento, temos dois exemplos na figura 15, a agregação simples mostra que um Carro é composto por Pneus, todavia, um objeto pneu existiria sem necessariamente termos o objeto Carro instanciado. Já no segundo exemplo, não teria muito sentido existir um capítulo de um livro sem que esteja relacionado ao livro, uma vez que o capítulo é uma parte do livro, ou melhor, caso um objeto livro seja removido, os seus capítulos serão deletados conjuntamente, ou seja, a existência do todo depende das partes, nesse caso teríamos a agregação por composição. A diferença no diagrama entre as duas é que o losango é preenchido ou não depende do relacionamento.

Figura 15 – Exemplo de agregação simples e agregação por composição



Fonte: Elaborada pelo autor.

E aí, vamos para o código! Para facilitar o entendimento da agregação, mostramos abaixo a implementação da agregação simples e da agregação por composição. Para agregação simples, teríamos duas classes separadas, uma vez que o objeto Pneu poderia existir sem o Carro.

```
1.    public class Carro {
2.    private String modelo;
3.    private Pneu[] pneus;
4.    ...
5.    // ver a implementação da classe no circuito
anterior
6.  }
7.  public class Pneu {
8.    private String marca;
9.    private float calibragem;
11.   ...
12. }
13. public class TestCarro {
14.  public static void main(String args[]) {
15.       Carro c1 = new Carro();
16.       int qtePneus = c1.getQtdPneus();
17.       ...
18.       Pneu pneu1 = new Pneu();
19.       c1.incluirPneu(pneu1);
20.       c1.trocarPneu();
21.   }
22.  }
```

Na implementação da classe Carro (linha 1), vemos que temos um objeto da classe Pneu, ou melhor, um vetor de objetos, uma vez que podemos ter várias instâncias de Pneus em objeto carro (linhas 15 e 18). Como a agregação é simples, faz sentido termos duas classes separadas e independentes. Seria possível, por exemplo, criar um objeto da classe Pneu sem ter uma objeto da classe Carro.

```
1.    public class Livro {
2.    private String modelo;
3.    private Capitulo[] capitulos;
4.    private int qtePaginas;
5.    ...
6.  // implementação da classe interna Capitulo
7.  static class Capitulo {
8.    private int qtePaginas;
9.    private String titulo;
11.   private String assunto;
12.   ...
13. }
13. public Livro(int qteCapitulos, String titulo, ...) {
14.       capitulos = new Capitulos[qteCapitulos];
15.       this.titulo = titulo;
16.       ...
17.  }
18.  public Capitulo getCapitulo(int n_capitulo) {
19.       return capitulos[n_capitulo];
20.   }
21.   // ...
22.  }
```

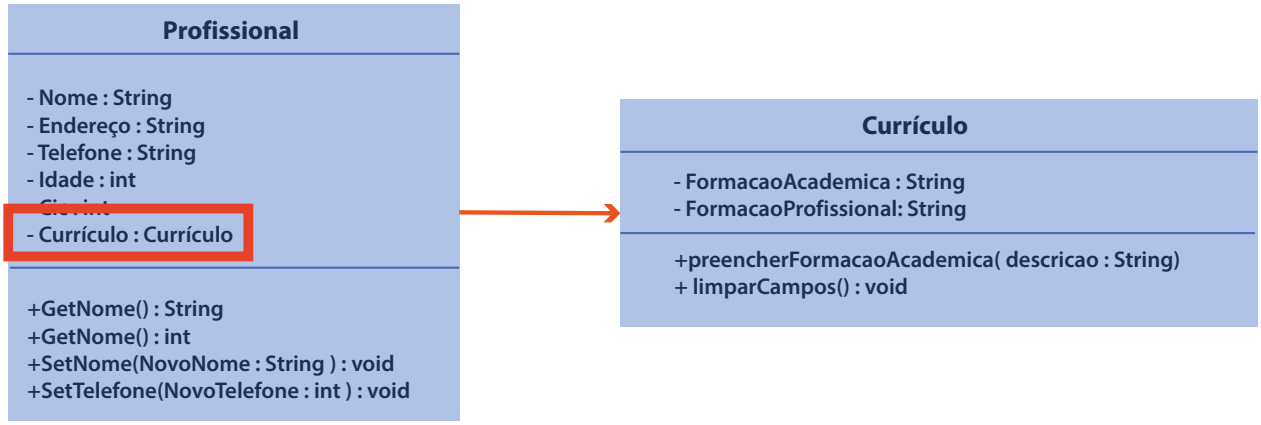
Na implementação da classe Livro, temos um exemplo de implementação de agregação por composição, em que a classe capítulo é uma classe interna do Java, ou seja, só existe “internamente” na classe Livro (linha 7).

Atenção!

A diferença entre associação binária, agregação simples e agregação por composição é muito tênue, como se pode verificar no exemplo da figura 16, em que, na definição da classe Profissional, temos um objeto da classe Currículo. Resta a interpretação de que se um objeto da classe currículo pode existir sem um objeto Profissional, hmm, isso vai depender do contexto do sistema e o que se quer representar no mesmo. Lembra da abstração, um dos pilares estudado?

Nesse caso, poderíamos definir essa associação como simples como também por agregação, dependendo do contexto do problema e da modelagem.

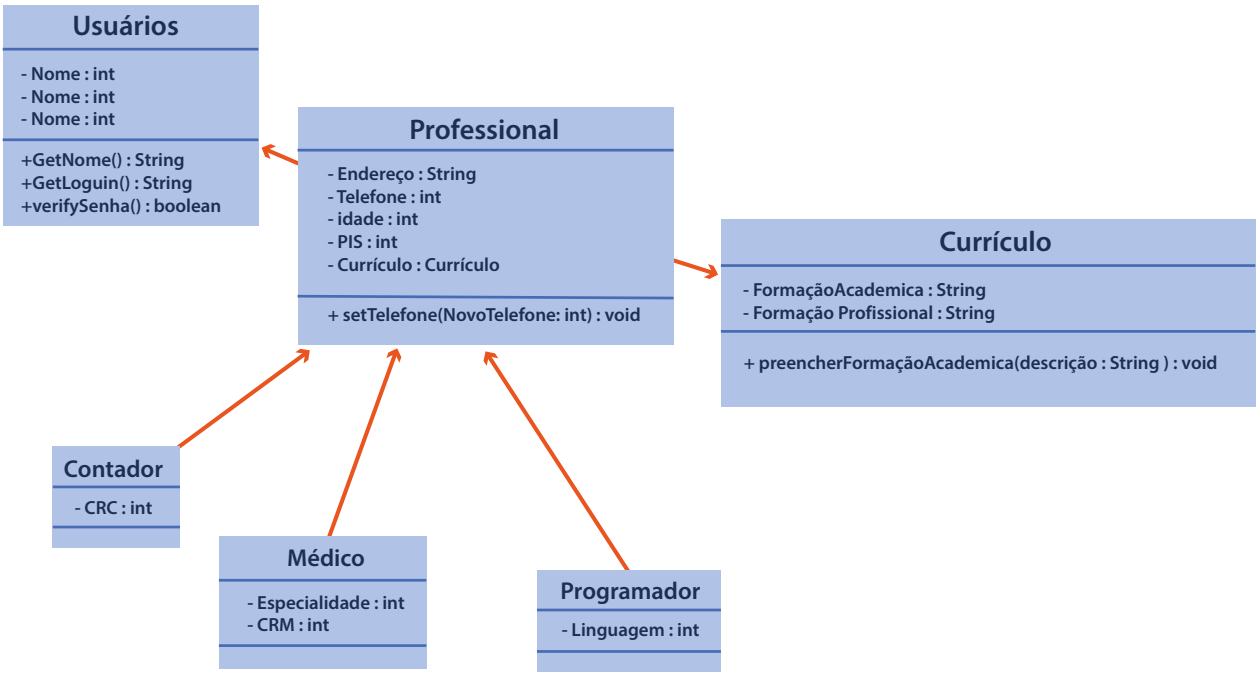
Figura 16 – Associação simples



Fonte: Notas 2019.

Temos ainda a associação de especialização e generalização, como podemos ver na figura 17, cuja classe profissional é uma especialização da classe Usuário e, por sua vez, as classes Programador, Médico e Contador são especializações da classe Profissional. No sentido contrário da seta, temos a generalização, ou seja, a classe Profissional é uma generalização para a classe Programador.

Figura 17 – Associação por especialização e generalização



Fonte: Elaborada pelo autor.

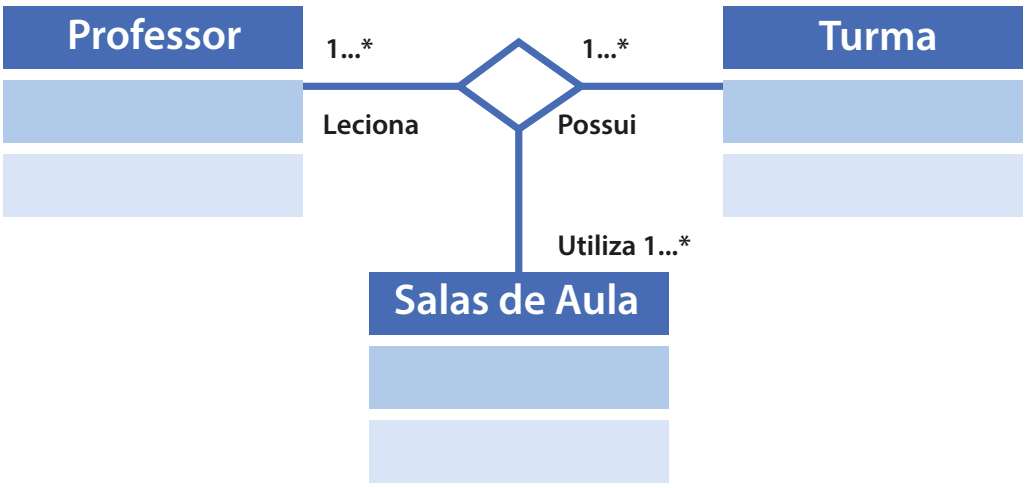
Esse tipo de relacionamento é o derivado da herança, cujo sentido do relacionamento "É UM". Na figura 17, um profissional "é um" usuário, médico "é um" Profissional; já o relacionamento de agregação tem um sentido de "TEM UM". Um profissional tem um currículo, um livro é composto por capítulos e um carro tem pneus, mostrando só exemplos trabalhados neste texto. Pense um pouco! Nos objetos sobre os quais trabalhamos no nosso dia a dia, quais teriam associação simples, por composição ou por agregação simples.

Importante!

Existem, também, a associação **ternária** e **N-ária**, na qual podemos ter um relacionamento entre várias classes. A associação ternária é mais comum, como se pode ver na figura 18. Na associação N-ária, o modelo do sistema pode ficar muito complexo e não ser usual. Nesse caso, seria melhor tentar quebrar o relacionamento entre mais classes. Para elucidar melhor a associação ternária, temos o exemplo da figura 18, na qual um professor pode lecionar em 1 ou mais turmas e utilizar de 1 ou mais salas de aula para isso. Ou seja, a sala de aula é alocada para combinação professor/turma, e não só para uma delas de forma particular.

Observe, também, os estereótipos "Leciona", "Possui", "Utiliza" que identificam o relacionamento entre essas classes. Como foi mencionado acima, é comum existir uma "confusão" entre os diagramas de classes e os diagramas modelo entidade e relacionamento – famoso MER, que são utilizados para representar o modelo lógico da base de dados. Contudo, a relação entre os dois diagramas realmente é muito próxima, uma vez que os dados que mapeamos no banco são semelhantes àqueles que mapeamos em memória, que, no caso do POO, é armazenado nos objetos.

Figura 18 – Associação ternária



Fonte: GUEDES, 2006.

Por fim, outro diagrama importante é o de objetos, uma vez que ele mostra o relacionamento entre os objetos e os seus dados em um determinado momento. Ele representa uma instância específica de um diagrama de classes em um determinado momento e, visualmente, você verá muitas semelhanças entre os dois diagramas.

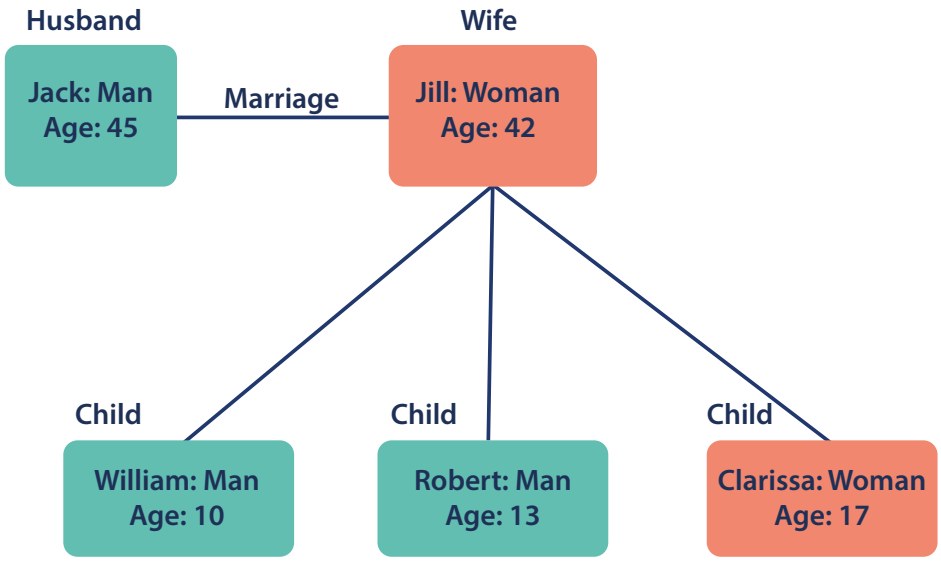
Na figura 19, temos um diagrama de objetos mostrando o "marido", a "esposa" e "os filhos" com seus atributos populares.

EXEMPLO:

O "marido" é Jack, de 45 anos de idade, sexo: masculino. Já a esposa é a Jill, idade 42 anos, e do sexo feminino.

Neste Circuito de Aprendizagem, aprendemos o relacionamento entre as classes e objetos, uma parte essencial no paradigma orientado a objetos, uma vez que a forma como esses objetos vão trocar mensagens e se comunicar é que viabiliza a implementação de um sistema computacional.

Figura 19 – Diagrama de Objetos



Fonte: Lucidchart, 2021.

Voltar ao sumário

Resumo:

Iniciamos nesta unidade o estudo da programação orientada a objetos, exemplificando-o por meio de casos comuns do nosso dia a dia e mostrando o poder da abstração e do encapsulamento na representação dos sistemas. Analisamos os 4 pilares da OO: abstração, encapsulamento, polimorfismo e herança, cada qual com suas peculiaridades para facilitar a representação e a modelagem dos problemas.

Em seguida, mostramos as vantagens do paradigma OO em relação ao estruturado ou procedural, mostrando as vantagens de se representar os problemas dessa forma. Exemplificamos a implementação do POO com a linguagem Java, reconhecidamente uma das principais, senão, a principal linguagem de programação nesse paradigma, que constitui uma forma de nos comunicarmos com o computador seguindo determinadas regras.

Identificamos também os tipos de relacionamento entre as classes – associação, agregação, composição e especialização. Apresentamos como representar os modelos orientados a objetos através dos diagramas UML, sendo o principal o diagrama de classe. Além disso, aprendemos os principais estereótipos da linguagem UML e como implementá-los na linguagem Java.

Referências

CAELUM, Curso de Java e Orientação a objetos básica, 2021. Disponível em: https://www.caelum.com.br/apostila-java-orientacao-objetos/orientacao-a-objetos-basica.

CESCHIM, Beatriz; GANIKO-DUTRA, Matheus; CALDEIRA, Ana Maria de Andrade, 2020. Disponível em: https://www.scielo.br/j/ciedu/a/ dfxhQ5MtSkSHfFBshpXsJyJ/?lang=pt

CONSULTA, Medicas, 2021. Disponível em: https://consultaremedios.com.br/ floratil.

DELPHI, Active, 2021; Disponível em: http://www.activedelphi.com.br/cyber.php.

EDELWEISS, Nina ; LIVI, Maria Aparecida Castro. Algoritmos e programação com exemplos em Pascal e C. Porto Alegre: Bookman, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/9788582601907.(DIGITAL) (Cód.:612)

FEBRAFAR, Redes associadas 2021, Disponível em: https://www.febrafar.com.br / manual-de-analise-de-insumos-farmaceuticos-disponivel/various-pills-and-capsules-in-container/.

FOWLER, Martin. UML essencial: um breve guia para linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2011. Disponível em: https://integrada.minhabiblioteca.com.br/books/9788560031382. (DIGITAL)

(Cód.:6903)

FLORENZANO, Claudio, Programação Orientada a Objetos: Uma introdução, 2021. Disponível em: https://www.cbsi.net.br/2014/02/programacao-orientada-objetos-uma.html

GRANCURSOS, 2021. Disponível em https://blog-static.infra.grancursosonline.com.br/wp-content/uploads/2020/06/UML.png.

HORSTMANN, Cay. Conceitos de computação com Java. 5. ed. Porto Alegre: Bookman, 2009. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788577804078. (DIGITAL) (Cód.:2813)

LUCIDCHART. O que é um diagrama de objetos, 2021. Disponível em: https://www.lucidchart.com/pages/pt/o-que-e-diagrama-de-objetos-uml .

MANZANO, José Augusto N. G. . Programação de computadores com Java. São Paulo: Érica, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/9788536531137. (DIGITAL) (Cód.:30259)

NOTAS, Notas de Aulas, 2019.

SEBESTA, Robert W. Conceitos de linguagens de programação. 4. ed. Porto Alegre: Bookman, 2000.

SCHILDT, Herbert. Java para iniciantes: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788582603376. (DIGITAL) (Cód.:6269)

SOUZA, Marco A. Furlan; GOMES, Marcelo Marques; SOARES, Márcio Vieira; CONCILIO, Ricardo. **Algoritmos e lógica de programação: um texto introdutório para a engenharia.** 3ed. São Paulo: Cengage Learning, 2019

STN, Tesouro Nacional, SICONFI: Sistema de Informações Contábeis e Fiscais do Setor Público Brasileiro, 2021. Disponível em: https://siconfi.tesouro.gov.br/siconfi.

USINAGEM: o que é e qual a sua importância. **www.ccvindustrial.com.br**, 2019. Disponível em: <https://ccvindustrial.com.br/usinagem-o-que-e-e-qual-a-sua-importancia>. Acesso em: 22 de dez. De 2020.

WAZLAWICK, Raul Sidnei. Análise e design orientados a objetos para sistema de informação : modelagem com UML, OCL e IFML. 3. ed. Rio de Janeiro: Elsevier, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/ 9788595153653/cfi/6/2!/4/2@0.00:0.00. (DIGITAL) (Cód.:30192)

WINDER, Russel; GRAHAM, Roberts (autor). Desenvolvendo software em Java. 3. ed. Rio de Janeiro: LTC, 2009. Disponível em: https://integrada.minhabiblioteca.com. br/books/978-85-216-1994-9. (DIGITAL) (Cód.:3517)

TIOBE, Index for september 2021, Disponível em: https://www.tiobe.com/tiobe-index/.

PROGRESSIVA, Classes e Objetos em JavaScript, 2021, disponível em: https://www.javascriptprogressivo.net/2019/01/Classe-Objeto-Propriedade-Metodos-JS.html

TREINAMENTOS, Apostilha de Orientação a Objetos em Java, 2021. Disponível em http://www.inf.furb.br/~marcel/k19-k11-orientacao-a-objetos-em-java.pdf.

WIKI, Programação Orientada a Objeto: Classes e Objetos, 2021. Disponível em: https://pooperrotti.fandom.com/pt-br/wiki/Classes_e_Objetos.

Voltar ao sumário

AUTOR
MAIKOL MAGALHÃES RODRIGUES

Possui graduação em Ciência da Computação pela Universidade Estadual do Ceará (1998) e mestrado em Ciência da Computação pela Universidade Estadual de Campinas (2001). Atualmente é analista de sistemas - Serviço Federal de Processamento de Dados, professor assistente da Faculdade Farias Brito e professor assistênte da Universidade de Fortaleza. Tem experiência na área de Ciência da Computação, com ênfase em Modelos Analíticos e de Simulação, atuando principalmente nos seguintes temas: programação linear inteira, programação matemática, otimização combinatória, softwares de otimização e modelos de programação linear.

RESPONSABILIDADE TÉCNICA

UNIVERSIDADE DE FORTALEZA ENSINANDO E APRENDENDO
VRE Vice-Reitoria de Ensino de Graduação e Pós-Graduação
Universidade de Fortaleza | EAD

COORDENAÇÃO DA EDUCAÇÃO A DISTÂNCIA

Coordenação Geral de EAD
Douglas Royer

Coordenação de Ensino e Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Ensino e Aprendizagem
Carla Dolores Menezes de Oliveira

Supervisão de Planejamento Educacional
Ana Flávia Beviláqua Melo

Supervisão de Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Operações e Atendimento
Mírian Cristina de Lima

Analista Educacional
Lara Meneses Saldanha Nepomuceno

Projeto Instrucional
Igor Gomes Rebouças

Revisão Gramatical
José Ferreira Silva Bastos

Identidade Visual / Arte
Francisco Cristiano Lopes de Sousa

Editoração / Diagramação
Régis da Silva Pereira

Produção de Áudio e Vídeo
Pedro Henrique de Moura Mendes

Programação / Implementação
Francisco Weslley Lima