

# Conceitos de Programação Orientada a Objetos – OOP

(Object-Oriented Programming)  
Fevereiro - 2007

## Índice

1. História .....	4
2. Introdução a Programação com Objetos .....	4
3. Conceitos Básicos .....	5
Abstração .....	5
Encapsulação .....	5
Compartilhamento .....	6
4. Modelo de Objetos e Definição de Classe .....	7
Objetos e Classes .....	7
Ligações e Associações .....	10
Agregação .....	11
5. Herança .....	12
6. Interface .....	14
7. Classe Abstrata .....	15
8. Escopo de classe, variável e método .....	16
9. Orientação a Objetos no ADVPL .....	17
Definição .....	17
Dicas úteis .....	17
Modelagem das classes de Exemplo .....	18
Criação de uma classe .....	19
Herança com ADVPL .....	20
Exemplo da criação de um objeto .....	20
10. Sugestões de Desenvolvimento .....	21
11. Referência Bibliográfica: .....	21

Histórico de Alterações		
Nome	Data	Observações
Marcelo Dolis Malatesta	23/02/2007	Criação
Michel Willian Mosca	23/02/2007	Criação

## 1. História

Nos anos 60 é lançado a Simula-67 que apresentou pela primeira vez os conceitos de classes, rotinas correlatas e subclasses. Na década de 70 a Seros-PARC cria a Smalltak até hoje considerada a mais pura das LPOO. No final da década de 80 aparece a C++, uma linguagem híbrida.

Orientação a Objetos é o maior avanço em software destes últimos anos. É uma forma mais natural de se analisar o mundo. Ela nos permite construir sistemas melhores e, além disso, de maneira mais fácil. Será a mais importante das tecnologias emergentes na área de software nos anos 90.

As técnicas estruturadas que, sem dúvida, atualmente são as mais populares na comunidade de informática, obtiveram grande aceitação desde que foram lançadas no final dos anos 70. Contudo a medida que foram sendo utilizadas, a decomposição funcional mostrou-se inadequada em situações de sistemas complexos e principalmente para profissionais iniciantes. Os aperfeiçoamentos introduzidos em 1984, por Sthephen M e John F Palmer e, mais tarde, complementados por Stephen Mellor e Paul Ward, para sistemas em real-time, ajudaram a análise estruturada a se tornar mais eficiente. Contudo os sistemas criados com as técnicas estruturadas ainda são difíceis de serem incrementados com novas funções e as alterações em funções já existentes, muitas vezes, provocam sérios problemas em outras partes do software.

## 2. Introdução a Programação com Objetos

Um *objeto* é uma entidade do mundo real que tem uma *identidade*. Objetos podem representar entidades concretas (um arquivo no meu computador, uma bicicleta) ou entidades conceituais (uma estratégia de jogo, uma política de escalonamento em um sistema operacional). Cada objeto ter sua identidade significa que dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características.

Embora objetos tenham existência própria no mundo real, em termos de linguagem de programação um objeto necessita um mecanismo de identificação. Esta *identificação de objeto* deve ser única, uniforme e independente do conteúdo do objeto. Este é um dos mecanismos que permite a criação de coleções de objetos, as quais são também objetos em si.

A estrutura de um objeto é representada em termos de *atributos*. O comportamento de um objeto é representado pelo conjunto de *operações* que podem ser executadas sobre o objeto. Objetos com a mesma estrutura e o mesmo comportamento são agrupados em *classes*. Uma classe é uma abstração que descreve propriedades importantes para uma aplicação e simplesmente ignora o resto.

Cada classe descreve um conjunto (possivelmente infinito) de objetos individuais. Cada objeto é dito ser uma *instância* de uma classe. Assim, cada instância de uma classe tem seus próprios valores para cada atributo, mas dividem os nomes dos atributos e métodos com as outras instâncias da classe. Implicitamente, cada objeto contém uma referência para sua própria classe -- em outras palavras, ele sabe o que ele é.

*Polimorfismo* significa que a mesma operação pode se comportar de forma diferente em classes diferentes. Por exemplo, a operação *move* quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um *método* é uma implementação específica de uma operação para uma certa classe.

Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.

No mundo real, alguns objetos e classes podem ser descritos como casos especiais, ou *especializações*, de outros objetos e classes. Por exemplo, a classe de computadores pessoais com processador da linha 80x86 é uma especialização de computadores pessoais, que por sua vez é uma especialização de computadores. Não é desejável que tudo que já foi descrito para computadores tenha de ser repetido para computadores pessoais ou para computadores pessoais com processador da linha 80x86.

*Herança* é o mecanismo do paradigma de orientação a objetos que permite compartilhar atributos e operações entre classes baseada em um relacionamento hierárquico. Uma classe pode ser definida de forma genérica e depois refinada sucessivamente em termos de *subclasses* ou *classes derivadas*. Cada subclasse incorpora, ou *herda*, todas as propriedades de sua *superclasse* (ou *classe base*) e adiciona suas propriedades únicas e particulares. As propriedades da classe base não precisam ser repetidas em cada classe derivada. Esta capacidade de fatorar as propriedades comuns de diversas classes em uma superclasse pode reduzir dramaticamente a repetição de código em um projeto ou programa, sendo uma das principais vantagens da abordagem de orientação a objetos.

### 3. Conceitos Básicos

A abordagem de orientação a objetos favorece a aplicação de diversos conceitos considerados fundamentais para o desenvolvimento de bons programas, tais como abstração e encapsulação. Tais conceitos não são exclusivos desta abordagem, mas são suportados de forma melhor no desenvolvimento orientado a objetos do que em outras metodologias.

#### **Abstração**

*Abstração* consiste de focalizar nos aspectos essenciais inerentes a uma entidade e ignorar propriedades "acidentais." Em termos de desenvolvimento de sistemas, isto significa concentrar-se no que um objeto é e faz antes de se decidir como ele será implementado. O uso de abstração preserva a liberdade para tomar decisões de desenvolvimento ou de implementação apenas quando há um melhor entendimento do problema a ser resolvido.

Muitas linguagens de programação modernas suportam o conceito de abstração de dados; porém, o uso de abstração juntamente com polimorfismo e herança, como suportado em orientação a objetos, é um mecanismo muito mais poderoso.

O uso apropriado de abstração permite que um mesmo modelo conceitual (orientação a objetos) seja utilizado para todas as fases de desenvolvimento de um sistema, desde sua análise até sua documentação.

#### **Encapsulação**

*Encapsulação*, também referido como *esconder informação*, consiste em separar os aspectos externos de um objeto, os quais são acessíveis a outros objetos, dos detalhes internos de implementação do objeto, os quais permanecem escondidos dos outros objetos. O uso de

encapsulação evita que um programa torne-se tão interdependente que uma pequena mudança tenha grandes efeitos colaterais.

O uso de encapsulação permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que usam este objeto. Motivos para modificar a implementação de um objeto podem ser por exemplo melhoria de desempenho, correção de erros e mudança de plataforma de execução.

Assim como abstração, o conceito de encapsulação não é exclusivo da abordagem de orientação a objetos. Entretanto, a habilidade de se combinar estrutura de dados e comportamento em uma única entidade torna a encapsulação mais elegante e mais poderosa do que em linguagens convencionais que separam estruturas de dados e comportamento.

### **Compartilhamento**

Técnicas de orientação a objetos promovem compartilhamento em diversos níveis distintos. Herança de estrutura de dados e comportamento permite que estruturas comuns sejam compartilhadas entre diversas classes derivadas similares sem redundância. O compartilhamento de código usando herança é uma das grandes vantagens da orientação a objetos. Ainda mais importante que a economia de código é a clareza conceitual de reconhecer que operações diferentes são na verdade a mesma coisa, o que reduz o número de casos distintos que devem ser entendidos e analisados.

O desenvolvimento orientado a objetos não apenas permite que a informação dentro de um projeto seja compartilhada como também oferece a possibilidade de reaproveitar projetos e código em projetos futuros. As ferramentas para alcançar este compartilhamento, tais como abstração, encapsulação e herança, estão presentes na metodologia; uma estratégia de reuso entre projetos é a definição de bibliotecas de elementos reusáveis. Entretanto, orientação a objetos não é uma fórmula mágica para alcançar reusabilidade; para tanto, é preciso planejamento e disciplina para pensar em termos genéricos, não voltados simplesmente para a aplicação corrente.

#### 4. Modelo de Objetos e Definição de Classe

Um modelo de objetos busca capturar a estrutura estática de um sistema mostrando os objetos existentes, seus relacionamentos, e atributos e operações que caracterizam cada classe de objetos. É através do uso deste modelo que se enfatiza o desenvolvimento em termos de objetos ao invés de mecanismos tradicionais de desenvolvimento baseado em funcionalidades, permitindo uma representação mais próxima do mundo real.

Uma vez que as principais definições e conceitos da abordagem de orientação a objetos estão definidos, é possível introduzir o *modelo de objetos* que será adotado ao longo deste texto. O modelo apresentado é um subconjunto do modelo OMT (Object Modeling Technique), proposto por Rumbaugh e outros<sup>1,4</sup>. OMT também introduz uma representação diagramática para este modelo, a qual será também apresentada aqui.

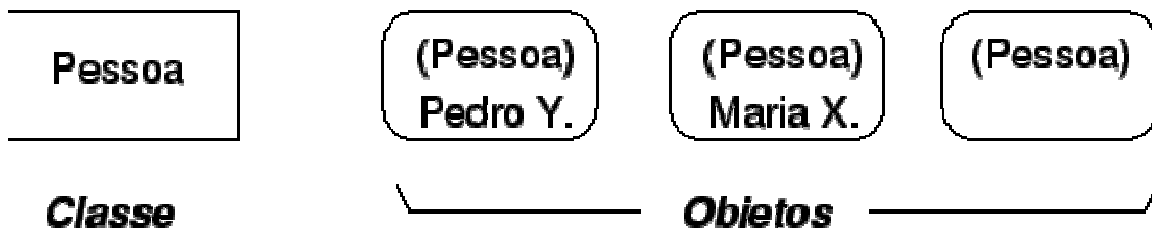
##### **Objetos e Classes**

*Objeto* é definido neste modelo como um conceito, abstração ou coisa com limites e significados bem definidos para a aplicação em questão. Objetos têm dois propósitos: promover o entendimento do mundo real e suportar uma base prática para uma implementação computacional. Não existe uma maneira "correta" de decompor um problema em objetos; esta decomposição depende do julgamento do projetista e da natureza do problema. Todos objetos têm identidade própria e são distinguíveis.

Uma *classe de objetos* descreve um grupo de objetos com propriedades (atributos) similares, comportamento (operações) similares, relacionamentos comuns com outros objetos e uma semântica comum. Por exemplo, *Pessoa* e *Companhia* são classes de objetos. Cada pessoa tem um nome e uma idade; estes seriam os atributos comuns da classe. Companhias também podem ter os mesmos atributos nome e idade definidos. Entretanto, devido à distinção semântica elas provavelmente estariam agrupados em outra classe que não *Pessoa*. Como se pode observar, o agrupamento em classes não leva em conta apenas o compartilhamento de propriedades.

Todo objeto sabe a que classe ele pertence, ou seja, a classe de um objeto é um atributo implícito do objeto. Este conceito é suportado na maior parte das linguagens de programação orientada a objetos, tais como C ++.

OMT define dois tipos de diagramas de objetos, diagramas de classes e diagramas de instâncias. Um diagrama de classe é um *esquema*, ou seja, um padrão ou gabarito que descreve as muitas possíveis instâncias de dados. Um diagrama de instâncias descreve como um conjunto particular de objetos está relacionado. Diagramas de instâncias são úteis para apresentar exemplos e documentar casos de testes; diagramas de classes têm uso mais amplo. A Figura apresenta a notação adotada para estes diagramas.



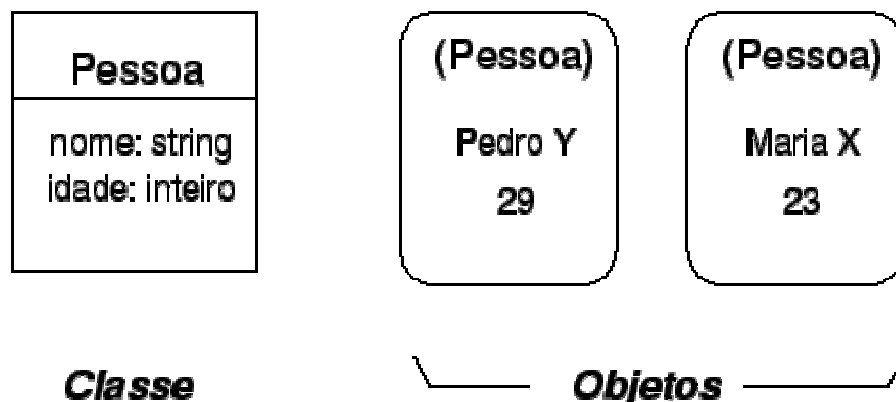
**Figura:** Representação diagramática de OMT para classes e objetos. Um diagrama de classe é apresentado à esquerda. Um possível diagrama de instâncias é apresentado à direita.

## Atributos

Um atributo é um valor de dado assumido pelos objetos de uma classe. Nome, idade e peso são exemplos de atributos de objetos *Pessoa*. Cor, peso e modelo são possíveis atributos de objetos *Carro*. Cada atributo tem um valor para cada instância de objeto. Por exemplo, o atributo *idade* tem valor "29" no objeto *Pedro Y*. Em outras palavras, Pedro Y tem 29 anos de idade. Diferentes instâncias de objetos podem ter o mesmo valor para um dado atributo.

Cada nome de atributo é único para uma dada classe, mas não necessariamente único entre todas as classes. Por exemplo, ambos *Pessoa* e *Companhia* podem ter um atributo chamado *endereço*.

No diagrama de classes, atributos são listados no segundo segmento da caixa que representa a classe. O nome do atributo pode ser seguido por detalhes opcionais, tais como o tipo de dado assumido e valor *default*. A Figura 1 mostra esta representação.



**Figura:** Representação diagramática de OMT para classes e objetos com atributos. Um diagrama de classe com atributos é apresentado à esquerda. Um possível diagrama de instâncias com os respectivos valores é apresentado à direita.

Não se deve confundir identificadores internos de objetos com atributos do mundo real. Identificadores de objetos são uma conveniência de implementação, e não têm nenhum significado para o domínio da aplicação. Por exemplo, CIC e RG não são identificadores de objetos, mas sim verdadeiros atributos do mundo real.

## Operações e Métodos

Uma operação é uma função ou transformação que pode ser aplicada a ou por objetos em uma classe. Por exemplo, *abrir*, *salvar* e *imprimir* são operações que podem ser aplicadas a objetos da classe *Arquivo*. Todos objetos em uma classe compartilham as mesmas operações.


Toda operação tem um objeto-alvo como um argumento implícito. O comportamento de uma operação depende da classe de seu alvo. Como um objeto "sabe" qual sua classe, é possível escolher a implementação correta da operação. Além disto, outros argumentos (parâmetros) podem ser necessários para uma operação.

Uma mesma operação pode se aplicar a diversas classes diferentes. Uma operação como esta é dita ser *polimórfica*, ou seja, ela pode assumir distintas formas em classes diferentes.



Um *método* é a implementação de uma operação para uma classe. Por exemplo, a operação *imprimir* pode ser implementada de forma distinta, dependendo se o arquivo a ser impresso contém apenas texto ASCII, é um arquivo de um processador de texto ou binário. Todos estes métodos executam a mesma operação -- imprimir o arquivo; porém, cada método será implementado por um diferente código.

A *assinatura* de um método é dada pelo número e tipos de argumentos do método, assim como por seu valor de retorno. Uma estratégia de desenvolvimento recomendável é manter assinaturas coerentes para métodos implementando uma dada operação, assim como um comportamento consistente entre as implementações.


Em termos de diagramas OMT, operações são listadas na terceira parte da caixa de uma classe. Cada nome de operação pode ser seguida por detalhes opcionais, tais como lista de argumentos e tipo de retorno. A lista de argumentos é apresentada entre parênteses após o nome da operação. Uma lista de argumentos vazia indica que a operação não tem argumentos; da ausência da lista de argumentos não se pode concluir nada. O tipo de resultado vem após a lista de argumentos, sendo precedido por dois pontos (:). Caso a operação retorne resultado, este não deve ser omitido -- esta é a forma de distingui-la de operações que não retornam resultado. Exemplos de representação de operações em OMT são apresentados na Figura .

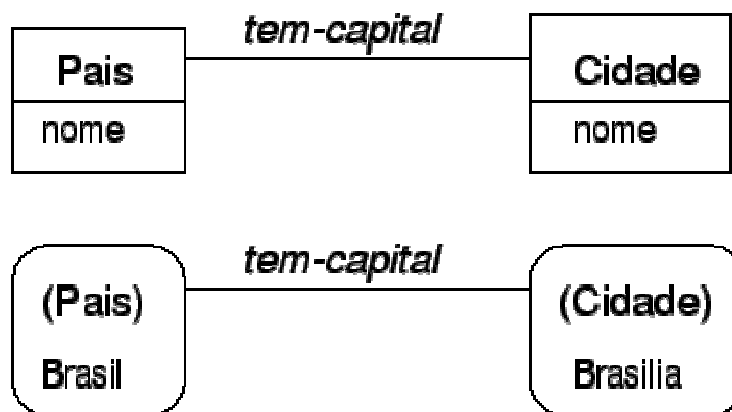
<b>Pessoa</b>
nome: string idade: inteiro
muda-empr muda-ender

<b>Objeto Geometrico</b>
cor posicao
move (delta: Vetor) select (p: Ponto): Boolean


## Ligações e Associações

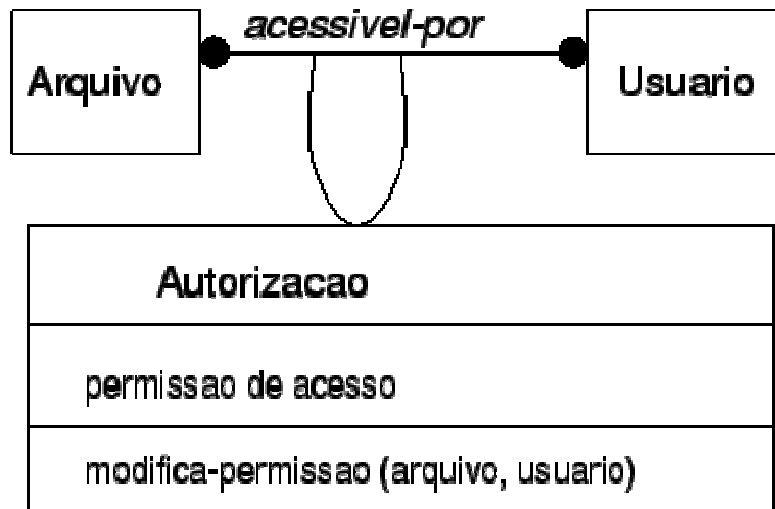
Ligações e associações são os mecanismos para estabelecer relacionamentos entre objetos e classes. Uma ligação é uma conexão física ou conceitual entre duas instâncias de objetos. Por exemplo, Pedro Y *trabalha-para* Companhia W. Uma ligação é uma instância de uma associação. Uma associação descreve um grupo de ligações com estrutura e semântica comuns, tal como ``uma pessoa *trabalha-para* uma companhia." Uma associação descreve um conjunto de ligações potenciais da mesma forma que uma classe descreve um conjunto de objetos potenciais.

A notação de diagramas OMT para associação é uma linha conectando duas classes. Uma ligação é representada como uma linha conectando objetos. Nomes de associações são usualmente apresentada em itálico. Se entre um par de classes só existe uma única associação cujo sentido deva ser óbvio, então o nome da associação pode ser omitido. A Figura  apresenta um exemplo de diagrama OMT com associações.



**Figura:** Representação diagramática de OMT para associações entre classes (topo) e ligações entre objetos (abaixo).

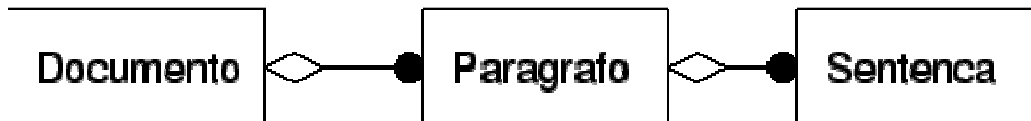
Alguns atributos podem dizer respeito a associações, e não a classes. Para tais casos, OMT introduz o conceito de *atributo de ligação*. Quando a associação tem ainda operações associadas, então ela pode ser modelada como uma classe que está ``conectada" à associação. Um exemplo deste caso é apresentado na Figura .



**Figura:** Representação diagramática de OMT para associações entre classes com atributos. Neste caso, os atributos da associação estão representados através de uma classe explícita, *Autorização*. O círculo preto no final da linha da associação indica que mais de um objeto de uma classe podem estar associados a cada objeto da outra classe. Um círculo vazado indicaria que possivelmente nenhum objeto poderia estar associado, ou seja, o conceito de associação opcional.

### Agregação

Uma *agregação* é um relacionamento do tipo "uma-parte-de," nos quais objetos representando os componentes de alguma coisa são associados com objetos representando uma *montagem*. Por exemplo, o texto de um documento pode ser visto como um conjunto de parágrafos, e cada parágrafo é um conjunto de sentenças (Figura 1.10).



**Figura:** Representação diagramática de OMT para agregação.


Agregação é uma forma de associação com alguma semântica adicional. Por exemplo, agregação é *transitiva*: se *A* é parte de *B* e *B* é parte de *C*, então *A* é parte de *C*. Agregação também é *anti-simétrica*: se *A* é parte de *B*, então *B* não é parte de *A*.

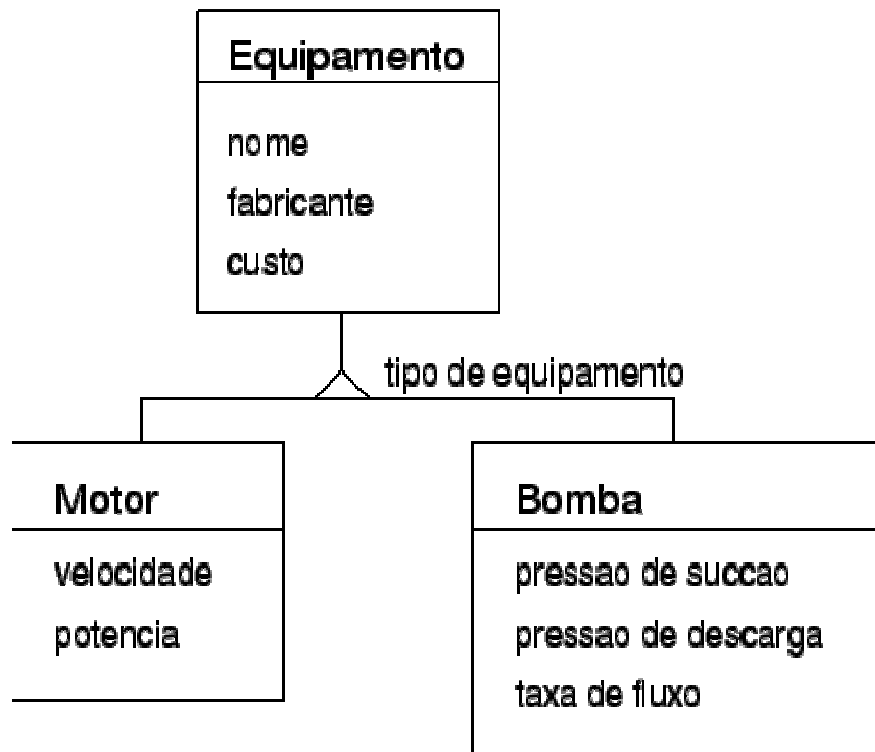
## 5. Herança

Generalização e herança são abstrações poderosas para compartilhar similaridades entre classes e ao mesmo tempo preservar suas diferenças.

*Generalização* é o relacionamento entre uma classe e um ou mais versões refinadas (especializadas) desta classe. A classe sendo refinada é chamada de superclasse ou *classe base*, enquanto que a versão refinada da classe é chamada uma subclasse ou *classe derivada*. Atributos e operações comuns a um grupo de classes derivadas são colocadas como atributos e operações da classe base, sendo compartilhados por cada classe derivada. Diz-se que cada classe derivada *herda* as características de sua classe base. Algumas vezes, generalização é chamada de relacionamento *is-a* (é-um), porque cada instância de uma classe derivada é também uma instância da classe base.

Generalização e herança são transitivas, isto é, podem ser recursivamente aplicadas a um número arbitrário de níveis. Cada classe derivada não apenas herda todas as características de todos seus ancestrais como também pode acrescentar seus atributos e operações específicos.

A Figura  mostra a notação diagramática de OMT para representar generalização, um triângulo com o vértice apontado para a classe base. Um *discriminador* pode estar associado a cada associação do tipo generalização; este é um atributo do tipo enumeração que indica qual a propriedade de um objeto está sendo abstraída pelo relacionamento de generalização. Este discriminador é simplesmente um nome para a base de generalização.



**Figura:** Representação diagramática de OMT para generalização.

Uma classe derivada pode *sobrepor*<sup>1,2</sup> uma característica de sua classe base definindo uma característica própria com o mesmo nome. A característica local (da classe derivada) irá refinar e substituir a característica da classe base. Uma característica pode ser sobreposta, por exemplo, por questões de refinamento de especificação ou por questões de desempenho.

Entre as características que podem ser sobrepostas estão valores *default* de atributos e métodos de operação. Uma boa estratégia de desenvolvimento não deve sobrepor uma característica de forma inconsistente com a semântica da classe base.

## 6. Interface

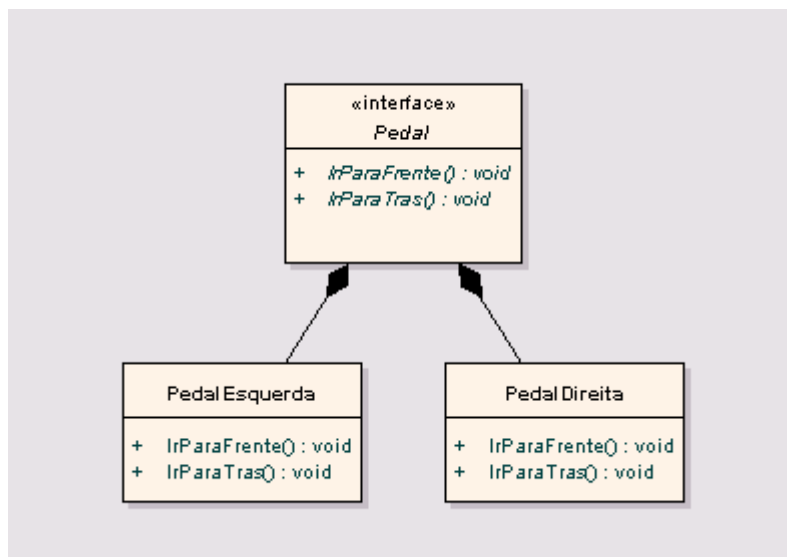
É um tipo definido da qual não se deseja que sejam criadas instâncias da mesma, mas apenas deve ser seguida como padrão para a criação de novas classes como um “contrato”.

Na interface são apenas definidas as assinaturas de operações e ou propriedades que deverão ser suportadas pelas classes que utilizem a interface, sem a inclusão de código de programação na interface.

O uso de interfaces esta diretamente associada à troca de mensagens entre os objetos, desta maneira o programador cria qualquer nova classe que necessite e assim mesmo as mensagens continuam sendo trocadas entre objetos sem prejuízos ao sistema.

As classes que implementam uma interface, devem implementar todas as operações definidas pela interface e o respectivo código de programação para que sempre que um objeto necessitar trocar uma mensagem não cause erros.

Veja o exemplo abaixo:



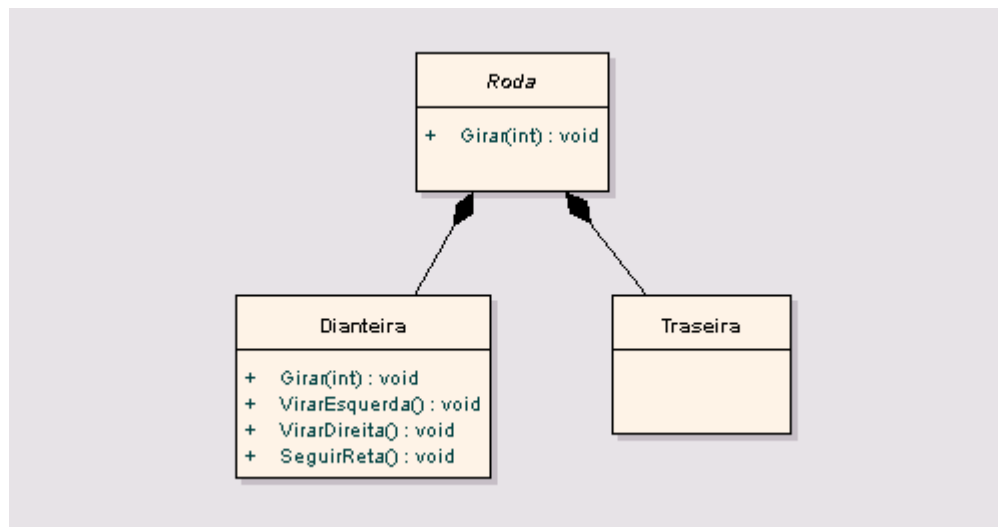
**Figura:** Exemplo de interface

No exemplo acima, é demonstrado o uso de interface com o caso dos pedais de uma bicicleta onde o pedal da esquerda e o pedal da direita executam os mesmos movimentos, então, respondem às mesmas mensagens.

Há uma interface chamada **Pedal** que descreve duas operações *IrParaFrente()* e *IrParaTrás()* e duas classes que implementam a interface Pedal sendo as classes **PedalEsquerda** e **PedalDireita** que deverão responder as mensagens definidas na interface.

## 7. Classe Abstrata

A classe abstrata é um tipo de dado bastante similar à **Interface** onde também são definidas as assinaturas de operações e propriedades não podendo ser instanciado diretamente por uma classe, servindo apenas ao propósito de ser um padrão para a criação de novas classes como um contrato. A grande diferença da Classe Abstrata para a Interface é a possibilidade de apenas definir as assinaturas das operações, deixando a responsabilidade da implementação do código da operação para a classe que implemente a Classe Abstrata como ocorre com a Interface, ou então também implementar o código de implementação para a operação na própria Classe Abstrata podendo o código ser sobrescrito pela classe implementadora respeitando-se a assinatura original da operação.



**Figura:** Exemplo de Classe Abstrata

No exemplo acima, há a classe abstrata **Roda** e as classes que a implementam **Dianteira** e **Traseira**. Neste caso a classe **Traseira** utiliza as operações como foram implementadas na classe **Roda**, já a classe **Dianteira** sobrescreveu a operação **Girar** para melhor atender ao seu propósito.

## 8. Escopo de classe, variável e método

Em tempo de desenvolvimento, o programador/ analista deve escolher o escopo de uma classe, variável ou método que nada mais é como a definição de como será a visibilidade perante outros objetos.

Falando em OOP podemos ter os seguintes modelos de escopo:

- **Public** : todos os objetos visualizam.
- **Protected** : apenas os métodos da mesma classe ou classes que recebem herança da classe são capazes de visualizar.
- **Private**: apenas os métodos da mesma classe são capazes de visualizar.
- **Package**: pode ser visualizado por todas as classes que compartilham do mesmo pacote.



## 9. Orientação a Objetos no ADVPL

### Definição

Classes no Protheus podem conter métodos, propriedades e herança, conforme a sintaxe abaixo:

```
CLASS <NOME_DA_CLASSE> [FROM <NOME_DA_CLASSE_HERDADA>]  
    //Declarando uma propriedade/variável  
    DATA <NOME_DA_PROPRIEDADE>  
  
    //Declarando o método que é executado quando a classe é criada  
    METHOD NEW() CONSTRUCTOR  
  
    //Declarando o método que não necessita ser implementado pela classe. Utilizado por  
    interface.  
    METHOD <NOME_DO_MÉTODO>(PARAM1, PARAMN) VIRTUAL  
  
    //Declarando um método da classe  
    METHOD <NOME_DO_MÉTODO>(PARAM1, PARAMN)  
  
ENDCLASS
```

O método de inicialização é executado sempre que uma nova instância da classe for criada pelo sistema e sempre deve retornar a própria classe.

```
METHOD NEW() CLASS <NOME_DA_CLASSE>  
    //Coloque as propriedades e funções que devem ser executadas na inicialização  
RETURN SELF
```

A criação de métodos de uma classe é realizada da mesma maneira que já estamos acostumados a criar funções em ADVPL.

```
METHOD <NOME_DO_MÉTODO>(PARAM1, PARAMN) CLASS <NOME_DA_CLASSE>  
    //Coloque o código de programação da função.  
RETURN
```

### Dicas úteis

**//Use para acessar propriedades e métodos da classe**

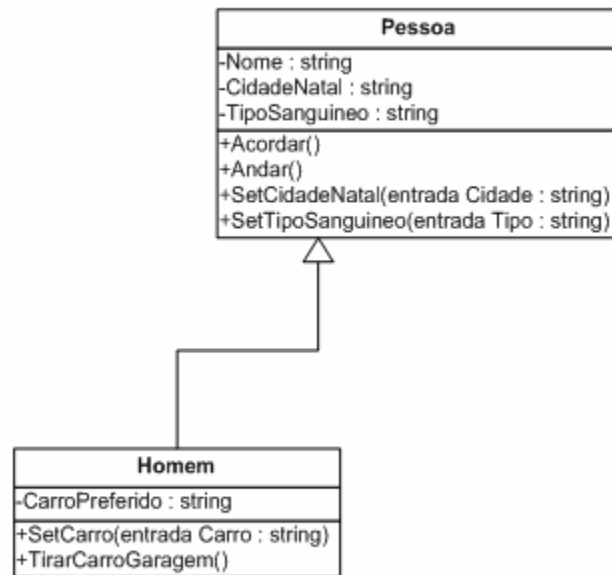
```
::  
ou  
:Self
```

**//Use para acessar propriedades e métodos da classe herdada**

```
:Parent
```

**Importante:** Para maiores detalhes sobre Classes e Objetos no ADVPL veja o arquivo de ajuda do Norton Guides chamado FW19.NG.

**Modelagem das classes de Exemplo**



### *Criação de uma classe*

```
Class Pessoa
    Data Nome
    Data CidadeNatal
    Data TipoSanguineo

    Method New(Nome) Constructor
    Method Acordar ()
    Method Andar ()
    Method SetCidadeNatal (Cidade)
    Method TipoSanguineo (Tipo)

EndClass

Method New(Nome) Class Pessoa
    ::Nome := Nome
Return Self

Method Acordar () Class Pessoa
    conout ("Bom Dia!!")
Return

Method Andar () Class Pessoa
    conout ("Andando")
Return

Method SetCidadeNatal (Cidade) Class Pessoa
    ::CidadeNatal := Cidade
Return

Method TipoSanguineo (Tipo) Class Pessoa
    ::TipoSanguineo := Tipo
Return
```

### ***Herança com ADVPL***

```
Class Homem From Pessoa
    Data CarroPreferido

    Method New(Nome) Constructor
    Method SetCarro(Carro)
    Method TirarCarroGaragem()
EndClass

Method New(Nome) Class Homem
    ::Nome := Nome
Return Self

Method SetCarro(Carro) Class Homem
    ::CarroPreferido := Carro
Return

Method TirarCarroGaragem() Class Homem
    conout("Tirando o carro da garagem")
Return
```

### ***Exemplo da criação de um objeto***

```
User Function CriaHomem()
Local oHomem := Homem():New("Roberto")

oHomem:SetCidadeNatal("Sao Paulo")
oHomem:TipoSanguineo("O-")
oHomem:SetCarro("Golf")
oHomem:Andar()
oHomem:TirarCarroGaragem()
conout("Nome da Pessoa:" + oHomem:Nome)

oHomem := NIL

Return
```

## 10. Sugestões de Desenvolvimento

Na construção de um modelo para uma aplicação, as seguintes sugestões devem ser observadas a fim de se obter resultados claros e consistentes:

1. Não comece a construir um modelo de objetos simplesmente definindo classes, associações e heranças. A primeira coisa a se fazer é entender o problema a ser resolvido.
2. Tente manter seu modelo simples. Evite complicações desnecessárias.
3. Escolha nomes cuidadosamente. Nomes são importantes e carregam conotações poderosas. Nomes devem ser descritivos, claros e não deixar ambiguidades. A escolha de bons nomes é um dos aspectos mais difíceis da modelagem.
4. Não "enterre" apontadores ou outras referências a objetos dentro de objetos como atributos. Ao invés disto, modele estas referências como associações. Isto torna o modelo mais claro e independente da implementação.
5. Tente evitar associações que envolvam três ou mais classes de objetos. Muitas vezes, estes tipos de associações podem ser decompostos em termos de associações binárias, tornando o modelo mais claro.
6. Não transfira os atributos de ligação para dentro de uma das classes.
7. Tente evitar hierarquias de generalização muito profundas.
8. Não se surpreenda se o seu modelo necessitar várias revisões; isto é o normal.
9. Sempre documente seus modelos de objetos. O diagrama pode especificar a estrutura do modelo, mas nem sempre é suficiente para descrever as razões por trás da definição do modelo. Uma explicação escrita pode clarificar pontos tais como significado de nomes e explicar a razão para cada classe e relacionamento.
10. Nem sempre todas as construções OMT são necessárias para descrever uma aplicação. Use apenas aquelas que forem adequadas para o problema analisado.

## 11. Referência Bibliográfica:

1. BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML Guia do usuário**. Rio de Janeiro : Campus, 2000.
2. DEITEL, H. M.; DEITEL, P. J.. **JAVA Como Programar**. Porto Alegre : Bookman, 2003.