2. Conceitos básicos e terminologia

Para que se possa utilizar uma linguagem de programação orientada a objetos é necessário primeiramente conhecer os conceitos e a terminologia utilizada por este paradigma.

2.1. Abstração de dados

Um conceito que teve muita influência na POO chama-se programação com tipos abstratos de dados. Tipos abstratos de dados possuem características como *encapsulamento* e *modularidade*.

A noção de tipos de dados ocorre na maioria das linguagens de programação tradicionais. Na declaração do tipo de uma variável, delimita-se o conjunto de valores que ela pode assumir e as operações que ela pode sofrer. A especificação de um tipo de dados deve definir os objetos constituintes do tipo e as operações aplicáveis a estes objetos. Além disso, é possível estabelecer uma maneira de representação para os objetos.

Geralmente, uma linguagem de programação provê alguns tipos básicos pré-definidos (tipos primitivos) e ainda oferece mecanismos para definição de novos tipos de dados renomeando tipos existentes ou agregando alguns tipos primitivos e/ou definidos pelo usuário (em Pascal, por exemplo, utiliza-se o comando *type* para criar novos tipos de dados). Existem linguagens que além de proporcionarem tipos de dados primitivos e tipos de dados definidos pelo usuário, incorporam o conceito de tipos abstratos de dados. Tipos abstratos de dados envolvem a disponibilidade de dados e operações (comportamento) sobre estes dados em uma única unidade.

A abstração de dados é utilizada para introduzir um novo tipo de objeto, que é considerado útil no domínio do problema a ser resolvido. Os usuários do tipo abstrato de dados preocupam-se apenas com o comportamento dos objetos do tipo, demonstrado em termos de operações significativas para tais objetos, não necessitando conhecer como estes objetos estão representados ou como as operações são realizadas neles (ocultamento de informação). Portanto, uma abstração de dados consiste de um conjunto de valores e de operações que completamente caracterizam o comportamento dos objetos. Esta propriedade é garantida fazendo-se com que as operações sejam a única maneira de criar e manipular os objetos. Como conseqüência, é necessário incluir operações suficientes para proporcionar todas as possíveis ações que os objetos possam sofrer.

Mais precisamente, um tipo abstrato de dados pode ser definido como um tipo de dados que satisfaz as seguintes condições:

- A representação e definição do tipo e as operações nos objetos do tipo são descritos em uma única unidade sintática (como em uma *Unit*, em Turbo Pascal)
- A representação (implementação) dos objetos do tipo é escondida das unidades de programa que utilizam o tipo; portanto, as únicas operações

possíveis em tais objetos são aquelas que fazem parte da definição (interface) do tipo.

O estilo de programação com tipos abstratos de dados inclui o princípio de **encapsulamento**, que proporciona ocultamento e proteção de informação, viabilizando a manutenção e facilitando a evolução de sistemas. Com o encapsulamento da estrutura de dados que representa os objetos e dos procedimentos que representam as possíveis operações sobre os objetos, tem-se que uma alteração na estrutura de dados provavelmente exigirá modificações nos procedimentos, porém o efeito dessas modificações fica restrito às fronteiras da unidade sintática que descreve o tipo (a *Unit*, no caso). Se a interface permanecer a mesma, as unidades de programa que utilizam o tipo não necessitam sofrer alterações.

Uma outra vantagem que pode ser citada é o considerável aumento de confiabilidade obtido através do princípio da proteção. Unidades de programa que utilizam um tipo não estão aptas a fazer modificações diretamente. Elas somente podem chamar as operações que estão disponíveis na interface, aumentando a integridade dos objetos.

2.2. Objetos

Na visão de uma linguagem imperativa tradicional (Pascal, C, COBOL, etc.), os *objetos* aparecem como uma única entidade autônoma que combina a representação da informação (estruturas de dados) e sua manipulação (procedimentos), uma vez que possuem capacidade de processamento e armazenam um estado local. Pode-se dizer que um objeto é composto de (Figura 2.1):

- *Propriedades*: são as informações, estruturas de dados que representam o estado interno do objeto. Em geral, não são acessíveis aos demais objetos.
- *Comportamento*: conjunto de operações, chamados de *métodos*, que agem sobre as propriedades. Os métodos são ativados (disparados) quando o objeto recebe uma *mensagem* solicitando sua execução. Embora não seja obrigatório, em geral uma mensagem recebe o mesmo nome do método que ela dispara. O conjunto de mensagens que um objeto está apto a receber está definido na sua *interface*.
- *Identidade*: é uma propriedade que diferencia um objeto de outro; ou seja, seu nome.

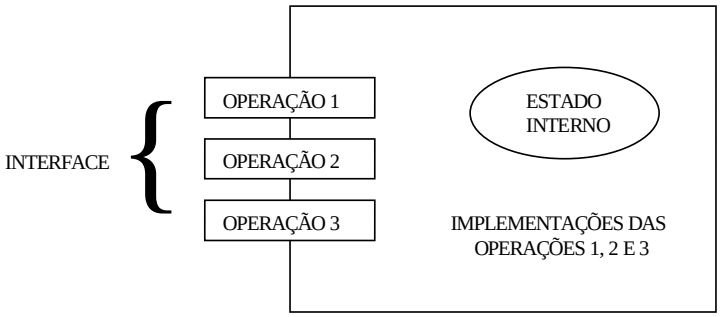

Figura 2.1. Estrutura de um objeto

Enquanto que os conceitos de dados e procedimentos são freqüentemente tratados separadamente nas linguagens de programação tradicionais, em POO eles são reunidos em uma única entidade: o objeto. A figura 2.2 apresenta outra visualização para um objeto.

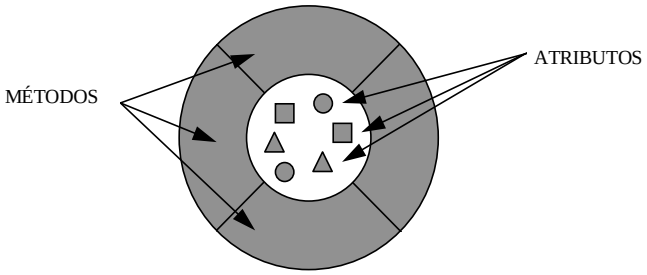

Figura 2.2. Outra representação para um objeto

No mundo real não é difícil a identificação de objetos (em termos de sistemas, objetos são todas as entidades que podem ser modeladas, não apenas os nossos conhecidos objetos inanimados). Como exemplo, em uma empresa pode-se identificar claramente objetos da classe (seção 2.4) *empregado*.

Um empregado possui uma identidade própria, seu nome: José da Silva. Possui também propriedades como: endereço, idade, dependentes, salário, cargo, entre outras. O comportamento pode ser determinado por operações como: aumentar salário, listar dependentes e alterar cargo. As propriedades somente podem ser manipuladas através das operações definidas na interface do objeto, de modo que a forma de armazenamento das propriedades e a implementação das operações são desconhecidas pelas outras entidades externas (encapsulamento de informações).

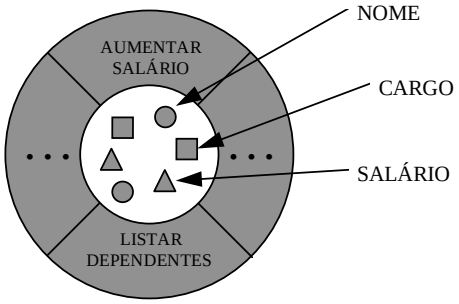

Figura 2.3. O objeto "Empregado"

Benefícios proporcionados pelos objetos:

Uma vez que objetos utilizam o princípio da abstração de dados (seção 2.1), o encapsulamento de informação proporciona dois benefícios principais para o desenvolvimento de sistemas:

- *Modularidade*: o código fonte para um objeto pode ser escrito e mantido independentemente da código fonte de outros objetos. Além disso, um objeto pode ser facilmente migrado para outros sistemas.

- *Ocultamento de informação*: um objeto tem uma interface pública que os outros objetos podem utilizar para estabelecer comunicação com ele. Mas, o objeto mantém informações e métodos privados que podem ser alterados a qualquer hora sem afetar os outros objetos que dependem dele. Ou seja, não é necessário saber como o objeto é implementado para poder utilizá-lo.

2.3. Mensagens

Um objeto sozinho não é muito útil e geralmente ele aparece como um componente de um grande programa que contem muitos outros objetos. Através da interação destes objetos pode-se obter uma grande funcionalidade e comportamentos mais complexos.

Objetos de software interagem e comunicam-se com os outros através de mensagens. Quando o objeto A deseja que o objeto B execute um de seus métodos, o objeto A envia uma mensagem ao objeto B (Figura 2.4). Algumas vezes o objeto receptor precisa de mais informação para que ele saiba exatamente o que deve fazer; esta informação é transmitida juntamente com a mensagem através de *parâmetros*.

Figura 2.4. O envio de mensagens entre objetos

Uma mensagem é formada por três componentes básicos:

- o objeto a quem a mensagem é endereçada (receptor)
- o nome do método que se deseja executar
- os parâmetros (se existirem) necessários ao método

Estes três componentes são informação suficiente para o objeto receptor executar a método desejado. Nada mais é necessário. Então, objetos em processos distintos ou ainda em máquinas distintas podem comunicar-se através do uso de mensagens.

Como exemplo, pode-se imaginar a realização de uma operação aritmética utilizando-se objetos, métodos e mensagens. Uma adição é realizada enviando-se uma mensagem a um objeto representando o número. A mensagem especifica que a operação desejada é a adição e também o número que deve ser somado ao objeto receptor. Então, a operação "*x*+ *y*" é interpretada como a mensagem "+" sendo enviada ao objeto "x" com o parâmetro "y". Pode parecer estranho à primeira vista, mas em SmallTalk operações aritméticas são realmente tratadas dessa forma.

A título de comparação com as linguagens convencionais, uma mensagem se comporta como uma chamada de subrotina (procedimento ou função). A sintaxe em geral também é muito parecida.

Benefícios proporcionados pelas mensagens:

- Desde que tudo que um objeto pode fazer é expressado pelos seus métodos, a troca de mensagens suporta todos os tipos possíveis de interação entre objetos
- Para enviar e receber mensagens, objetos não precisam estar no mesmo processo ou máquina (ex: CORBA)

2.4. Métodos

Um método implementa algum aspecto do comportamento do objeto. Comportamento é a forma como um objeto age e reage, em termos das suas trocas de estado e troca de mensagens.

Um método é uma função ou procedimento que é definido na classe e tipicamente pode acessar o estado interno de um objeto da classe para realizar alguma operação. Pode ser pensado como sendo um procedimento cujo primeiro parâmetro é o objeto no qual deve trabalhar. Este objeto é chamado receptor (seção 2.3). Abaixo é apresentada uma notação possível para o envio de uma mensagem (invocação do método)

receptor.nome_da_mensagem(par1, par2, par3)

Muitas vezes quando se trabalha com objetos deseja-se realizar certas operações no momento da criação e destruição de um objeto. Em algumas linguagens é dado um tratamento de modo a ficar transparente ao programador a sua implementação. Em outras, é necessário utilizar métodos especiais para este fim, chamados de **construtores** e **destrutores**.

Construtores são usados para criar e inicializar objetos novos. Tipicamente, a inicialização é baseada em valores passados como parâmetros para o construtor. Destrutores são usados para destruir objetos. Quando um destrutor é invocado, as ações definidas pelo usuário são executadas, e então a área de memória alocada para o objeto é liberada. Em algumas linguagens, como C++, o construtor é chamado automaticamente quando um objeto é declarado. Em outras, como Object Pascal, é necessário chamar explicitamente o construtor antes de poder utilizá-lo.

Um exemplo de utilização de construtores e destrutores seria gerenciar a quantidade de objetos de uma determinada classe que já foram criados até o momento. No construtor pode-se colocar código para incrementar uma variável e no destrutor o código para decrementá-la. Na seção 4 são apresentados exemplos da utilização de construtores e destrutores.

2.5. Classes

Objetos de estrutura e comportamento idênticos são descritos como pertencendo a uma classe, de tal forma que a descrição de suas propriedades pode ser feita de uma só

vez, de forma concisa, independente do número de objetos idênticos em termos de estrutura e comportamento que possam existir em uma aplicação. A noção de um objeto é equivalente ao conceito de uma variável em programação convencional, pois especifica uma área de armazenamento, enquanto que a classe é vista como um tipo abstrato de dados, uma vez que representa a definição de um tipo.

Cada objeto criado a partir de uma classe é denominado de *instância* dessa classe. Uma classe provê toda a informação necessária para construir e utilizar objetos de um tipo particular, ou seja, descreve a forma da memória privada e como se realizam as operações das suas instâncias. Os métodos residem nas classes, uma vez que todas as instâncias de uma classe possuem o mesmo conjunto de métodos, ou seja, a mesma interface.

Cada instância pertence a uma classe e uma classe pode possuir múltiplas instâncias. Devido ao fato de todas as instâncias de uma classe compartilharem as mesmas operações, qualquer diferença de respostas a mensagens aceitas por elas, é determinada pelos valores das variáveis de instância.

Em algumas linguagens, aparece também a definição de variáveis de classe. Variáveis de classe surgem quando se tem a visão de classes sendo manipuladas como objetos. Esta abordagem se torna útil ao se tentar manter classes contendo informação. Como exemplo, uma classe poderia armazenar o número de objetos que tenham sido instanciados da classe até um certo momento. Cita-se, ainda, a importância de se ter classes como objetos para modelar a entidade que recebe uma mensagem requisitando a criação de um novo objeto (como em Object Pascal).

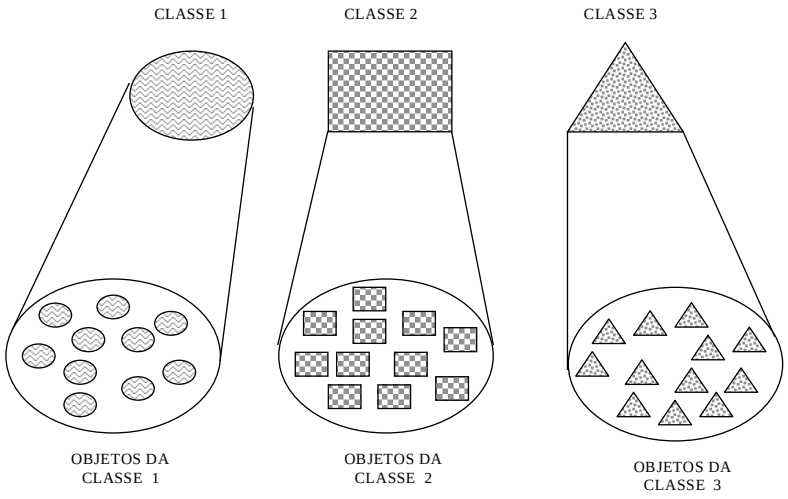

Figura 2.5. Relacionamento entre classes e objetos

A Figura 2.5 ilustra o relacionamento entre classes e objetos. Cada objeto instanciado a partir de uma classe possui as propriedades e comportamento definidos na classe, da mesma maneira que uma variável incorpora as características do seu tipo. A existência de classes proporciona um ganho em reusabilidade. pois o código das operações e a especificação da estrutura de um número potencialmente infinito de objetos estão definidos em um único local, a classe. Cada vez que um novo objeto é instanciado ou que uma mensagem é enviada, a definição da classe é reutilizada. Caso

não existissem classes, para cada novo objeto criado, seria preciso uma definição completa do objeto.

No exemplo de uma empresa seria possível criar classes para várias entidades a ser modeladas, como: empregados, departamentos, filiais, produtos, documentos fiscais, etc. Na definição da classe *empregado*, são especificados as suas propriedades e comportamento e escrito o código fonte da sua implementação. Quando um novo empregado é admitido na empresa, ele é criado como uma instância da classe *empregado*, e automaticamente possui todas as características da classe.

Metaclasses

Uma metaclasse é uma classe de classes. Pode-se julgar conveniente que, em uma linguagem ou ambiente, classes também possam ser manipuladas como objetos. Por exemplo, uma classe pode conter variáveis contendo informações úteis, como:

- o número de objetos que tenham sido instanciados da classe até certo instante;
- um valor médio de determinada propriedade, calculado sobre os valores específicos desta propriedade nas instâncias (por exemplo, média de idade de empregados).

Essas variáveis não devem ser criadas para cada objeto (instância), mas para cada classe deve existir apenas uma cópia de cada. Para implementar isto é conveniente considerar uma classe como sendo também um objeto. Neste caso, a classe precisará ser instância de outra classe. Esta classe de classes é chama de metaclasse.

Benefícios proporcionados pelas classes:

O maior benefício proporcionado pela utilização das classes é a reusabilidade de código, uma vez que todos os objetos instanciados a partir dela incorporam as suas propriedades e seu comportamento.

2.6. Herança

O mecanismo de herança permite a reutilização das propriedades de uma classe na definição de outra. A classe mais generalizada é chamada *superclasse* e a mais especializada, *subclasse*. No desenvolvimento de software, herança surge como a maneira para garantir que caixas pretas construídas não se tornem caixas isoladas. Através da herança, novas caixas pretas são obtidas aproveitando-se operações já implementadas, proporcionando a *reusabilidade de código* existente. Assim como no mundo dos seres humanos, o objeto descendente não tem nenhum trabalho para receber a herança. Ele a recebe simplesmente porque é um objeto descendente.

Herança é a contribuição original do paradigma de objetos, que o diferencia da programação com tipos abstratos de dados. Ela viabiliza a construção de sistemas a partir de componentes reusáveis. A herança não só suporta a reutilização entre sistemas, mas facilita diretamente a extensabilidade em um mesmo sistema. Diminui a quantidade de código para adicionar características a sistemas já existentes e também facilita a manutenção de sistemas, tanto provendo maior legibilidade do código existente, quanto diminuindo a quantidade de código a ser acrescentada.

A herança não é limitada a apenas um nível; a árvore de herança, ou *hierarquia de classes* pode ser tão profunda quanto for necessário. Métodos e variáveis internas são herdados por todos os objetos dos níveis para baixo. Quanto mais para baixo na hierarquia uma classe estiver, mais especializado o seu comportamento. Várias subclasses descendentes podem herdar as características de uma superclasse ancestral, assim como vários seres humanos herdam as características genéticas de um antepassado.

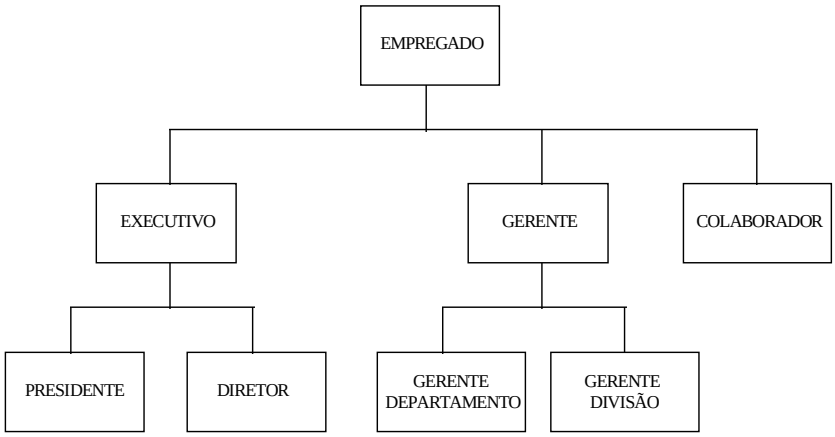

Figura 2.6. Herança na classe "Empregado"

Na figura 2.6 pode-se ver a hierarquia de classes para a classe "Empregado". As classes Executivo, Gerente e Colaborador herdam todos os métodos e variáveis da classe Empregado; além disso, podem redefini-los e criar novas características. A classe Presidente, também herda todas as características da classe Executivo, que por sua vez já herdou da classe Empregado.

A herança é interpretada da seguinte maneira: se B é subclasse de A (conseqüentemente, A é superclasse de B), então:

- os objetos da classe B suportam todas as operações suportadas pelos objetos da classe A, exceto aquelas redefinidas
- as variáveis de instância de B incluem todas as variáveis de instância de A.

Analisando os itens acima, tem-se que uma subclasse compartilha o comportamento e o estado de sua superclasse, podendo redefini-lo para seu próprio uso. Além disso, é possível acrescentar novas características que identificam um comportamento próprio. A maior parte das linguagens orientadas a objetos utiliza herança como uma técnica para prover suporte à *especialização*, e não permitem que uma classe exclua uma operação herdada.

A utilização de herança em linguagens orientadas a objetos pode afetar o encapsulamento de informação. Além dos clientes da classe que simplesmente requisitam operações sobre instâncias da classe, surge uma nova categoria de clientes que são as classes definidas através da herança, ou seja, as subclasses. Geralmente, às subclasses é permitido o total acesso à representação das variáveis de instâncias definidas na sua superclasse. Desta forma, compromete-se o princípio de encapsulamento, uma vez que as variáveis de instância de uma classe serão parte implícita da interface que ela proporciona para as suas possíveis subclasses. Mudanças na implementação de uma classe, como por exemplo a remoção de uma variável de instância, poderá comprometer

a implementação de suas subclasses. Assim, não existe mais a liberdade para modificar a representação de uma classe em provocar efeitos nos seus clientes.

Herança não necessariamente está limitada a uma única superclasse. Uma subclasse pode herdar características de diversas superclasses, introduzindo o conceito de *herança múltipla*. Indiscutivelmente, é obtido um maior potencial para o compartilhamento de código, porém a possibilidade de conflitos, como o confronto de nomes entre as operações herdadas, entre múltiplas superclasses aumenta a complexidade dos sistemas.

Benefícios proporcionados pela herança:

- Subclasses provêem comportamentos especializados tomando-se como base os elementos comuns definidos pela superclasse. Através do uso da herança, pode-se reutilizar o código da superclasse várias vezes.
- Pode-se implementar classes que definem comportamentos "genéricos" (chamadas de *classes abstratas*). A essência da superclasse é definida e pode ser parcialmente implementada, mas a maior parte da classe é deixada indefinida e não implementada. Os detalhes são definidos em subclasses especializadas, que podem ser implementadas inclusive por outros programadores de acordo com sua necessidade.

2.6. Polimorfismo

Polimorfismo refere-se à capacidade de dois ou mais objetos responderem à mesma mensagem, cada um a seu próprio modo. A utilização da herança torna-se fácil com o polimorfismo. Desde que não é necessário escrever um método com nome diferente para responder a cada mensagem, o código é mais fácil de entender. Por exemplo, sem polimorfismo, para inserir um novo empregado, seria necessário o seguinte código:

Colaborador1.***InsereColaborador***
Gerente1.***InsereGerente***
Presidente1.***InserePresidente***

Neste caso, Colaborador1, Gerente1 e Presidente1 são objetos das respectivamente das classes Colaborador, Gerente e Presidente. Com o polimorfismo, não é necessário ter um método diferente para cada tipo de objeto. Pode-se simplesmente escrever o seguinte:

Colaborador.***Insere***
Gerente.***Insere***
Presidente.***Insere***

Neste exemplo, os três diferente empregados têm três diferentes métodos para ser inseridos, embora o tenha sido utilizado o método com o mesmo nome. No código para esses empregados, seria necessário escrever três métodos diferentes: um para cada tipo de empregado. O mais importante a ser observado é que as três rotinas compartilham o mesmo nome.

O polimorfismo obviamente não nos auxilia a reutilizar o código. Em vez disso, ele é útil na compreensão de programas. Desde que uma grande quantidade de objetos compartilham o mesmo comportamento com implementações ligeiramente diferentes, encontrar uma maneira de diminuir o número de linhas de código para invocar estes comportamentos auxilia imensamente. Os programadores que são usuários da classe Empregado não precisam saber que a maneira de inserir um empregado é diferente; quando eles querem inserir, somente utilizam o método *Insere*.

Se considerarmos que o método *Insere* está sendo utilizado dentro de uma estrutura de repetição, fica bastante aparente o ganho obtido em legibilidade e tamanho do código. Sem a utilização de polimorfismo esta estrutura seria:

```
Para todos os Empregados Faça
        Se Empregado é Presidente Então
                Empregado[i].InserePresidente
        Se Empregado é Gerente Então
                Empregado[i].InsereGerente
        Se Empregado é Colaborador Então
                Empregado[i].InsereColaborador
        Fim Se
Fim Para
```

Com a utilização do polimorfismo seria possível utilizar o seguinte código:

```
Para todos os Empregados Faça
        Empregado[i].Insere
Fim Para
```

O polimorfismo visto nos exemplos acima implica na existência de métodos com implementação diferente com o mesmo nome em classes distintas. Outra forma simples de polimorfismo permite a existência de vários métodos com o mesmo nome, definidos na mesma classe, que se diferenciam pelo tipo ou número de parâmetros suportados. Isto é conhecido como *polimorfismo paramétrico*, ou *sobrecarga de operadores* (“overloading”). Neste caso, uma mensagem poderia ser enviada a um objeto com parâmetros de tipos diferentes (uma vez inteiro, outra real, por exemplo), ou com número variável de parâmetros. O nome da mensagem seria o mesmo, porém o método invocado seria escolhido de acordo com os parâmetros enviados.

Dependendo das características da linguagem e do tipo de polimorfismo suportado por ela, a implementação do polimorfismo pode exigir facilidades proporcionadas pela **ligação dinâmica**. De um modo geral, uma ligação (*binding*) é uma associação, possivelmente entre um atributo e uma entidade ou entre uma operação e um símbolo. Uma ligação é dita estática se ocorre em tempo de compilação ou de interligação (*linking*) e não é mais modificada durante toda a execução do programa. A ligação dinâmica é feita em tempo de execução ou pode ser alterada no decorrer da execução do programa.

Com a ligação dinâmica, a ligação do operador a uma particular operação pode ser feita em tempo de execução, isto é, o código compilado para um tipo abstrato de dados pode ser usado para diferentes tipos de dados, aumentando consideravelmente sua reusabilidade. Entretanto, o uso de ligação dinâmica compromete a eficiência de implementação.

Benefícios proporcionados pelo polimorfismo

- Legibilidade do código: a utilização do mesmo nome de método para vários objetos torna o código de mais fácil leitura e assimilação, facilitando muito a expansão e manutenção dos sistemas.
- Código de menor tamanho: o código mais claro torna-se também mais enxuto e elegante. Pode-se resolver os mesmos problemas da programação convencional com um código de tamanho reduzido.

2.7. Relações entre Objeto, Classe e Herança

Objetos, classes e o mecanismo de herança permitem a definição de hierarquias de abstrações, que facilitam o entendimento e o gerenciamento da complexidade dos sistemas estudados. Isto porque classes agrupam objetos com características iguais, enquanto herança estrutura classes semelhantes.

A capacidade em classificar objetos e classes concede grande poder de modelagem conceitual e classificação, podendo expressar relações entre comportamentos, tais como classificação/instanciação, generalização/especialização e agregação/composição. Facilita a compreensão humana do domínio estudado e também o desenvolvimento de programas que o satisfaça. Pode-se dizer que a grande vantagem do paradigma de objetos é a possibilidade de expressar diretamente este poder de modelagem conceitual numa linguagem de programação orientada a objetos. Assim, o modelo conceitual proposto durante a etapa de análise não se perde nas etapas de projeto e implementação. O que em geral ocorre é a sua extensão.

2.7.1. Classificação/Instanciação

A capacidade de classificar objetos (em classes) permite expressar relações do tipo classificação/instanciação. O relacionamento é feito a partir da observação de diversos fenômenos para categorização dos mesmos em grupos (classes), com base no conjunto de propriedades comuns a todos. Por exemplo, dois computadores, *IBM PC* e *Machintosh*, podem ser classificados como **instâncias** (objetos, modelos, ou espécimes) da classe (categoria) Microcomputador (Figura 2.7). A relação inversa é a de instanciação de uma publicação (*IBM PC*, por exemplo) a partir da classe Microcomputador.

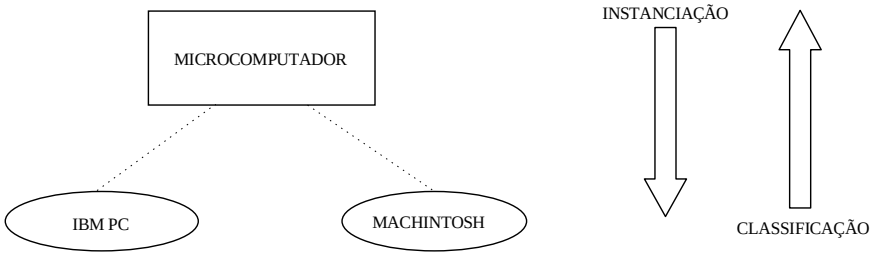

Figura 2.7. Relação de classificação/instanciação

2.7.2. Generalização/Especialização

Este tipo de relação ocorre quando, a partir a observação de duas classes, abstraímos delas uma classe mais genérica. Por exemplo, as classes Microcomputador e Mainframe podem ser considerados casos especiais da classe Computador. Esta classe

é considerada uma **generalização** das duas primeiras, que são chamadas de **especializações** da classe Computador (Figura 2.8). A idéia da generalização/especialização é a base para a classificação das espécies nas ciências naturais. Do ponto de vista de propriedades, o pressuposto é que as subclasses tenham todas as propriedades das classes de quem elas são especializações. Deve haver pelo menos uma propriedade que diferencie duas classes especializadas (subclasses) a partir da mesma classe genérica (superclasse). Este é o tipo de relação utilizado com o mecanismo de herança.

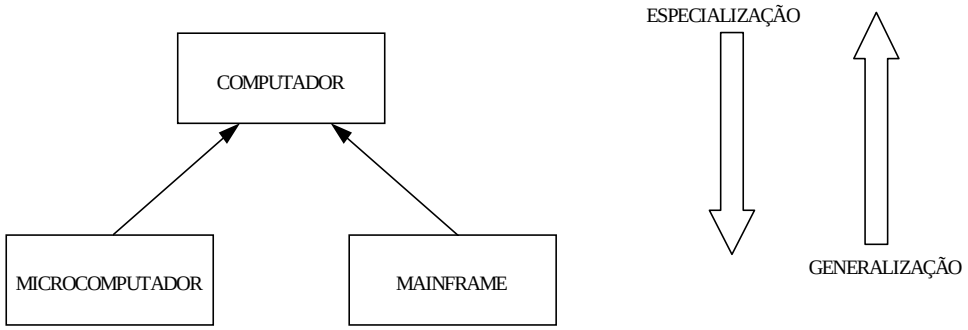

Figura 2.8. Relação de generalização/especialização

2.7.3. Composição/Decomposição

A relação de composição (ou agregação)/decomposição permite que objetos sejam compostos pela agregação de outros objetos ou componentes. Neste relacionamento, são determinados que instâncias (objetos) de uma classe são compostas por instâncias de outras classes. Essa operação é denominada **decomposição** e a relação inversa, a formação de uma nova classe como um agregado de classes preexistentes, é denominada **composição**. Por exemplo, instâncias da classe Microcomputador são compostas por, entre outras, instâncias das classes Teclado e Vídeo (Figura 2.9).

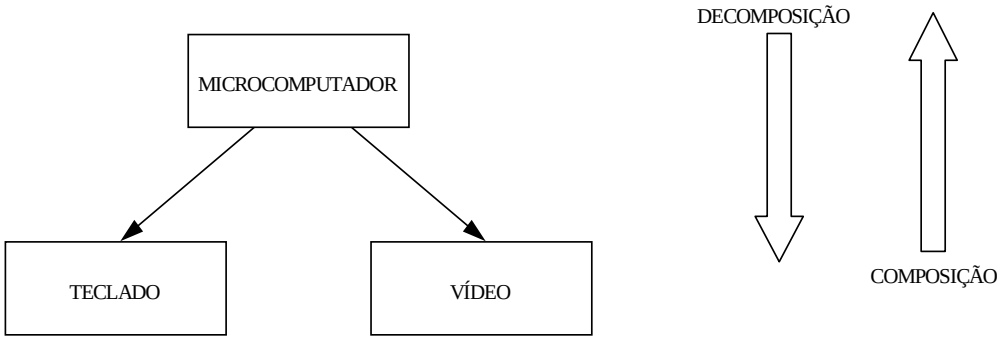

Figura 2.9. Relação de composição/decomposição