10

Programação Orientada Por Objetos

"For whatever we lose(like a you or a me)
it's always ourselves we find in the sea"

—e.e.cummings

O paradigma de programação orientada por objetos tem como princípio a solução de problemas pela cooperação de vários elementos, da mesma forma que usamos a prestação de serviço de outras pessoas para resolver vários dos nossos problemas.

A idéia de prestação de serviços pode ser ilustrada pelas situações diárias. Suponha que para a nossa pesquisa atual precisemos consultar um livro que não faça parte do acervo da biblioteca local. Então, requisitamos ao bibliotecário local que o dado livro seja reservado em outra biblioteca associada e trazido temporariamente para a biblioteca local. Quando requisitamos um serviço como este, não precisamos saber como o bibliotecário entrará em contato com as bibliotecas associadas, ou ainda de que forma fará a reserva e o transporte do livro. Precisamos apenas requisitar o serviço

à pessoa correta e recebê-lo, sem termos que participar ou interferir no serviço que está sendo prestado.

Note que na situação acima nos abstraímos da forma como o serviço foi executado e precisamos apenas fazer a **comunicação** com a pessoa (**objeto**) **responsável** pelo serviço e esperar pelo **resultado**. Essa idéia de prestação de serviço por agentes apropriados norteia o paradigma computacional de orientação a objetos. Esta tem como princípio a distribuição de serviços e responsabilidades de forma a diminuir a complexidade de cada um dos colaboradores.

As áreas de aplicação das linguagens orientadas a objetos têm-se expandido nos últimos anos. Essas linguagens foram criadas, juntamente com os seus ambientes, para dar suporte ao desenvolvimento de sistemas mais complexos (baseada em tipos abstratos), e por isso têm sido aplicadas em novos sistemas comeciais. O uso crescente dessas linguagens se dá principalmente pelo desenvolvimento de novas metodologias de sistemas que empregam o paradigma de orientação a objetos. A maioria dos sistemas atuais desenvolvidos para a Web são desenvolvidos nesse paradigma (na linguagem Java), dado o suporte à concorrência e agentes.

Neste capítulo abordamos, de forma superficial, os fundamentos inerentes à programação orientada por objetos, como os conceitos apresentados aparecem neste paradigma de programação e um exemplo de uma linguagem de programação orientada a objetos.

10.1 INTRODUÇÃO

Com o aumento da complexidade dos problemas computacionais a serem resolvidos, veio a necessidade de dividir as soluções computacionais em unidades menores, como vimos no Capítulo 7. As abstrações de processos e módulos conseguidas nas linguagens puramente imperativas não distribuem a responsabilidade dos estados dos programas para essas unidades (elas agrupam abstrações, mas é de responsabilidade do programa usá-las). Como conseqüência disso, os módulos não são unidades computacionalmente independentes. Daí, surgiu a idéia (com David Parnas, no início dos anos 70) de usar ocultação de informação como uma disciplina de programação para

que as unidades passassem a controlar os seus estados e os problemas pudessem ser resolvidos por colaboração.

Junto com a idéia de resolver problemas pela colaboração de vários agentes, os quais têm responsabilidade sobre os seus próprios serviços, surgiu a necessidade de prover recursos às linguagens de programação que dêem suporte a essa idéia. O conceito de programação orientada por objetos teve seu início com a linguagem Simula 67, mas teve um melhor alcance com Smalltalk[38] na década de 80. Alguns autores consideram até hoje Smalltalk como a única linguagem puramente orientada a objetos.

Sob o ponto de vista computacional devemos ter **objetos** que podem prover serviços sob a sua própria **responsabilidade**. Isso requer que os objetos sejam proprietários dos seus **estados** e possam dar **soluções próprias** aos serviços requisitados. As linguagens de programação orientadas a objetos permitem a criação de **objetos**, os quais são entidades computacionalmente ativas que guardam um conjunto de dados (os **atributos**), e os serviços (os **métodos**) que ele pode prover.

Para proporcionar a colaboração computacional dos objetos, deve-se também ter meios de fazer a **comunicação** entre eles (as **mensagens**). Então, uma vez que o objeto tem um serviço requisitado, denotado pela mensagem recebida, ele tem a responsabilidade de interpretar a mensagem, dentro do seu contexto, e executar o serviço requisitado.

Para o usuário do serviço, os passos necessários à sua execução não são relevantes, mas os efeitos, ou resultado, do mesmo. Contudo, o usuário deve escolher o objeto que possa fornecer o serviço e, para isso, os objetos devem estar classificados de forma que possam ser reconhecidos pelos usuários. As **classes** nas linguagens de programação agrupam os objetos com um mesmo comportamento definido, através de abstrações. Para fornecer um sistema de classificação, é necessário que tenhamos a relação entre as classes pela noção hierarquia junto com o mecanismo de **herança**.

A computação de um programa orientado a objetos pode ser visto como a ativação de objetos, mediante a sua criação, e colaboração entre esses objetos ativos. Como cada objeto é responsável pelos seus próprios dados (estado) e transformações (encapsulamento dos métodos), o estado do programa pode ser claramente visto como

o conjunto de estados dos objetos ativos. A solução computacional de um problema nesse paradigma pode ser reduzida a definir: quais objetos são necessários e como eles devem cooperar entre si. Assim, os objetos passam a ser unidades computacionais claramente independentes (com seus próprios dados e comportamento), e resolver o problema significa resolver cada uma dessas unidades juntamente com a comunicação entre elas.

As seções que seguem tratam os fundamentos das linguagens de programação orientadas por objetos à luz dos conceitos de linguagens vistos na Parte **??**. Em seguida apresentamos Samlltalk como linguagem exemplo.

10.2 FUNDAMENTOS

Os principais fundamentos das linguagens de programação com o objetivo de prover **objetos** estão centrados nos conceitos de abstrações. Cada objeto é um elemento abstrato responsável pelo seu estado e faz as transformações sobre tal estado mediante um conjunto fixo de regras de comportamento. A definição do conjunto de dados que um objeto deve conter (os atributos), bem como das regras de transformação sobre esses dados (os métodos) devem ser definidos como um tipo abstrato (as classes). Assim, a unidade de definição dos tipos abstratos são as classes e a ativação (ou uso) de um elemento daquele tipo é um objeto.

Dessa forma, cada objeto encapsula os seus dados e todas as suas possíveis transformações, enquanto as classes são o elemento de definição. Requisitar um serviço a um objeto significa enviar uma mensagem para ele com esta requisição, mas a forma como o objeto procede com o serviço fica oculto, os usuários não têm acesso. Com isso, o objeto possui completo controle sobre qualquer transformação do seu próprio estado, ele é o único que tem o poder de atuar sobre o seu estado, o que o caracteriza como unidade independente de computação. Apresentamos aqui as classes como elementos de definição de abstrações e os mecanismos existentes nas linguagens orientadas a objetos para expressar tais definições.

10.2.1 Classes e Métodos

Como visto no Capítulo 7, as classes são formas de definição de tipos abstratos. Os dados que um objeto deve conter são definidos pelos seus **atributos**, os quais devem guardar os valores necessários ao estado do objeto. As variáveis definidas nas classes que devem representar os estados dos objetos são as **variáveis de instância** e devem ser instanciadas para cada objeto de maneira a compor o estado do objeto. Como elas devem ser de propriedade de cada objeto, em algumas linguagens elas aparecem precedidas com uma palavra-chave que indica o escopo de visibilidade da mesma. Em C++ e Java a palavra `private` indica que as variáveis são de instância, em Smalltalk `InstanceVariableNames` é usada.

Exemplo 10.1 – Considere o tipo abstrato pilha de inteiros definido como uma classe na linguagem C++ (esquemático):

```
class pilha {
  private:            (variáveis de instância)
    int *ptr_pilha;
    int elemento;

  public:             (Métodos da Classe)
    pilha(){...}      (cria pilha)
    ~pilha(){...}     (destrói pilha)
    void vazia(){...}
    void insere(int elem){...}
    void retira(){...}
    int topo(){...}
}
```

As variáveis `*ptr_pilha` e `elemento` são as variáveis de instância dessa classe e cada objeto criado terá sua própria instância dessas variáveis. □

Algumas variáveis podem ser criadas em uma classe para serem acessadas por todos os objetos daquela dada classe. Isso é conveniente quando precisamos de uma forma de compartilhar informação entre objetos de uma mesma classe. Essas são as

chamadas **variáveis de classe**. Nas linguagens C++ e Java, são precedidas pela palavra `static`, enquanto em Smalltalk são precedidas por `classVariableNames`. Dentro da filosofia de orientação a objetos, essas variáveis devem ser usadas de forma restrita, apenas quando é necessário informação sobre uma classe, para que a independência entre os objetos de uma mesma classe seja preservada.

Além das variáveis de instância ou classe, devem ser definidos os **métodos**, os quais são os comportamentos permitidos para transformação dos dados abstratos definidos na classe. Objetos criados como instâncias de uma classe só podem responder a serviços definidos pelos métodos daquela classe. Alguns métodos podem aparecer como privativos à classe (`private` em C++), indicando que são serviços apenas internos (no exemplo acima mostramos apenas métodos públicos -`public`).

A criação de um objeto é dado por um `construtor` de objetos, os quais em algumas linguagens, aparecem com o nome da própria classe (`pilha` no exemplo acima) ou ainda com um nome determinado para a criação (`new`, em Smalltalk). A existência dos objetos dependem da sua criação nos programas, e vários objetos de uma mesma classe podem ser criados.

Exemplo 10.2 – Para o uso do tipo pilha, considere o seguinte trecho de programa:

```
void main(){
  int elem_int;
  pilha p1_int;       (1) (cria um objeto classe pilha)
  pilha p2_int;       (2) (cria um objeto classe pilha)
  ...
  p1_int.insere(9);   (3)
  ...
  if  (p2_int.vazia) ...
  ...
}
```

`p1_int` e `p1_int` são objetos do tipo `pilha`, de forma que são duas instâncias distintas da mesma classe. Note que cada mensagem é enviada diretamente ao obje-

to, `p1_int.insere(9)` por exemplo em (3), não havendo qualquer interferência sobre o estado do outro objeto criado (`p2_int`). □

A aplicação de uma operação sobre um objeto é na realidade uma requisição de serviço: o programa envia uma **mensagem** ao objeto, o qual atuará sobre o seu estado mediante o serviço requisitado. Alguns métodos atuam apenas sobre o estado do objeto, enquanto outros também passam um valor de **retorno** para o programa que requisitou o serviço (`topo`, no Exemplo 10.2, retorna um valor inteiro).

Dessa forma, um sistema orientado a objetos fornece os serviços necessários mediante a criação dos objetos necessários e comunicação entre eles pela passagem de mensagem.

10.2.2 Herança

As classes são definidas nas linguagens orientadas a objetos para agrupar objetos, e também para criar uma classificação entre os elementos definidos. Vários dos elementos que definimos podem estar relacionados com outros objetos. Em uma lista, por exemplo, temos uma seqüência de valores e nela podemos fazer operações sobre o primeiro e último elementos (consultar, inserir, retirar, etc). Uma pilha é também uma seqüência de valores, mas as operações só podem incidir sobre os valores do topo da pilha. Essa estruturas não são idênticas, mas existe claramente uma relação entre elas.

O sistema de classificação em orientação a objetos envolve uma hierarquia, e o mecanismo de **herança** foi criado para facilitar, sob o ponto de vista de definição, a relação de hierarquia entre as classes. Conceitualmente, uma classe criada como subclasse de outra herda todas as definições da classe superior (classe-pai), e de todas as outras superiores na hierarquia. Além disso, novos elementos podem ser criados na subclasse como, por exemplo, novos métodos e variáveis, ou ainda fazer uma sobrecarga dos elementos existentes.

Exemplo 10.3 – Considere a definição esquemática da classe que denota uma estrutura de lista de números inteiros em C++.

```
class lista {
```

```
  private:            (variáveis de instância)
    int *ptr_lista;
    int elemento;

  public:             (Métodos da Classe)
    lista(){...}      (cria lista)
    ~lista(){...}     (destrói lista)
    void vazia(){...}
    void inseretopo(int){...}
    void inserefim (int){...}
    void retiratopo(){...}
    void retirafim(){...}
    int topo(){...}
}
```

Uma estrutura de pilha é um uso específico de lista, onde as operações são realizadas apenas sobre elemento do topo como se verifica a seguir.

```
class pilha : private lista {
  public:                          (Métodos da Classe)
    pilha(){...}
    void vzia_pilha(){
         return lista : : vazia()};
    void insere_pilha(int elem){
         lista : : inseretopo(int elem)};
    void retira_pilha(){
         lista : : retiratopo()};
    int topo_pilha(){
         return lista : : topo()};

}
```

A classe `pilha` é definida como subclasse de `lista` de forma que as operações de inserção e remoção de elementos sejam restritas apenas à posição do topo da pilha. Todos os elementos definidos na classe são redefinições dos existentes na classe `lista` (`private` é usado para a classe). □

No exemplo acima é mostrado apenas um uso do mecanismo de herança para a hierarquia de classes. Timothy Budd [20] relata várias formas de como o mecanismo de herança pode ser usado na hierarquia de classes. Aqui, resumimos apenas os principais:

- especialização, especificação e extensão, onde a subclasse contém métodos redefinidos de forma compatível ou não implementados na classe-pai, mas mantém todos os elementos da classe-pai. Esta é a forma mais pura sob o ponto de vista de hierarquia de classes na orientação a objetos.

- generalização, limitação e construção, quando atributos e/ou métodos são inseridos ou ocultados à subclasse, ou ainda quando métodos são renomeados ou re-feitos nas subclasses.

- combinação ou uso de implementação, quando uma classe possui características de mais de uma classe-pai (herança múltipla), ou quando uma classe herda de outra apenas a implementação, apesar de não possuírem, conceitualmente, objetos relacionados (têm apenas algumas formas idênticas de transformar dados).

Os mecanismos de herança diferem entre as linguagens. Em Smalltalk, por exemplo, toda classe deve ter obrigatoriamente uma única classe-pai, não há herança múltipla, e os métodos podem ser sobrecarregados nas subclasses por redefinição. Em C++, as classes podem ser independentes, podem ter mais de uma classe-pai (herança múltipla), mas a sobrecarga de métodos só é admitida quando os parâmetros são equivalentes. Em Java, as classes que não têm a classe-pai designada são automaticamente subclasses da classe `Object` (classe-raiz) e a herança múltipla não é permitida diretamente (apenas pelas `interfaces`, as quais definem o protocolo, mas não a implementação).

10.2.3 Subtipos *versus* Subclasses

Uma pergunta sempre surge sobre o sistema de classificação das linguagens orientadas a objetos e o sistema de hierarquia de tipos. Podemos considerar que uma subclasse é um subtipo da sua classe-pai? A idéia de tipos e subtipos em linguagens de programação é usada sob diferentes perspectivas, desde uma visão do programador à do projetista de linguagens[66]. Da mesma forma, a noção de classes varia entre as linguagens. C++, por exemplo, teve sua definição de classes influenciada pela idéia de estrutura, e vista isoladamente preserva várias das noções aceitas sobre tipos. Quando o mecanismo de herança é considerado, a relação entre tipos e classes deve incorporar a noção de **capacidade de substituição**. Dado que uma instância da classe derivada é uma instância da classe-pai, uma variável do tipo derivado deve ser capaz de aparecer em qualquer lugar em que uma variável do tipo classe-pai é legal nos programas.

Vimos que os mecanismos de herança das linguagens atuais permitem que uma subclasse sobrecarregue métodos da classe-pai. Então, neste caso não há como garantir que o método sobrecarregado se comporte de maneira a preservar o comportamento do correspondente na classe-pai. Da mesma forma, quando limitações são permitidas para a construção de subclasses, a noção de subtipos também é falha.

Adele Goldberg, em [54], sugere que a relação entre subclasse e subtipo é possível quando algumas restrições são impostas à construção das subclasses: as subclasses podem apenas adicionar atributos e métodos, e as sobrecargas de métodos são permitidas quando não há erros de parâmetros e tipos e o comportamento da classe-pai é preservado. Timothy Budd[20], por outro lado, enfatiza a relação entre classes como objetos em vez de tipos, dadas as restrições impostas.

10.2.4 Polimorfismo e Vinculação Dinâmica

Um outro conceito importante em orientação a objetos é polimorfismo. Métodos definidos em uma classe são herdados pelas suas subclasses. Com isso, temos um tratamento uniforme (único método) para objetos definidos em classes diferentes. Para o uso do polimorfismo, as linguagens permitem que variáveis que representam objetos polimórficos sejam definidas, ou seja, variáveis que podem assumir tipos

relacionados. Para estas variáveis polimórficas, o objeto e métodos usados devem ser vinculados dinamicamente.

Na linguagem Smalltalk todas as variáveis são polimórficas porque as suas vinculações de tipos e métodos são dinâmicas. Em linguagens com vinculação estática de tipos, como C++ e Java, apenas algumas de sua variáveis são polimórficas. Em C++ apenas variáveis do tipo apontadores ou referências podem ser usadas como polimórficas. Em Java, as variáveis declaradas como objeto são polimórficas e por isso têm seus tipo e métodos vinculados dinamicamente.

Em geral, as vinculações estáticas de tipos, métodos e armazenamento melhoram a eficiência, enquanto a vinculação dinâmica de tipos e armazenamento aumentam a flexibilidade das linguagens.

A vinculação dinâmica de tipos requer que o objeto mantenha controle, inclusive, sobre o seu próprio tipo, designando uma completa responsabilidade ao objeto. Por outro lado, esse controle precisa ser realizado em tempo de execução para todas as operações que acessam o valor de dados, o que diminui a eficiência (apesar dos vários estudos já desenvolvidos). Para linguagens com vinculação dinâmica de tipos, as vinculações dos métodos também são dinâmicas, pois não podem ser decididos antes de que um tipo tenha sido vinculado. Isso também tem uma conseqüência na eficiência de processamento da linguagem.

Por essa razão, a maioria das linguagens implementa a vinculação estática de tipos quando este é declarado. Esse é o caso, por exemplo, das linguagens Java e C++, exceto para as suas variáveis polimórficas. A vinculação de métodos também pode ser estática, o que também torna a linguagem mais eficiente.

O polimorfismo é conseguido nas linguagens de programação orientada a objetos com as vinculações dinâmicas de tipos e métodos. Esta é uma questão de projeto de linguagens que devem priorizar a essência de orientação a objetos em detrimento da eficiência, ou o inverso. A vinculação estática de alguns elementos das linguagens não as descaracteriza como orientadas a objetos, mas deve ser claro aos programadores para quais elementos o polimorfismo é permitido na linguagem usada.

10.3 A LINGUAGEM SMALLTALK

A linguagem Smalltalk foi desenvolvida no Centro de Pesquisa da Xerox Palo Alto na década de 70 e teve algumas versões até chegar à versão Smalltalk-80, quando foi mais divulgada e conhecida. Ela tem como propósito de projeto ser uma linguagem puramente orientada a objetos. Assim, um programa Smalltalk é composto inteiramente por objetos que se comunicam entre si. Desde elementos tão simples quanto uma constante inteira a elementos mais sofisticados como acesso a arquivos, todos são objetos, proprietários dos seus próprios estados e formas de transformação sobre os seus dados. As classes na linguagem também são objetos, e a sua pureza de propósitos se reflete em um projeto elegante e uniforme de linguagem.

Nas seções que seguem, analisamos como alguns dos conceitos primordiais de linguagens de programação e, em particular, os fundamentos de orientação a objetos aparecem na linguagem Smalltalk. Essa não é uma análise exaustiva, mas apenas para elucidar alguns conceitos importantes e ajudar o leitor ao treino de reconhecer, na linguagem, o paradigma de orientação a objetos.

10.3.1 Objetos e Classes

Smalltalk possui um conjunto de objetos primitivos que são agrupados nas classes predefinidas da linguagem como: `Boolean, Character, Integer` e `Float`. Literais como 1 e 2 são considerados objetos da classe `Integer`, e estão preparados para responder a transformações como `+, -, *, //` (divisão de inteiros), `squared, even`, etc. Da mesma forma `true` e `false` são objetos da classe `Boolean`.

Os objetos compostos estão agrupados em classes como `Bag, Set, Dictionary, Interval, LinkedList, Array` e `String`. Cada uma dessas classes oferece um conjunto de métodos que manipulam esses dados.

Além dos objetos predefinidos na linguagem, podemos definir novas classes de objetos, as quais possuem quatro partes: o identificador da classe, o identificador da sua classe-pai, declaração das variáveis e declaração dos métodos. Os processadores de Smalltalk são baseados em ambientes gráficos, e a definição de cada uma dessas

partes é realizada em separado nos ambientes. Vale ressaltar que cada classe é também um objeto, permitindo que elas recebam mensagens, e que os métodos possam ser tanto de classes quanto de instâncias das classes. Isso faz com que todos os elementos sejam objetos, com o mesmo *status*. A criação de uma subclasse é realizada, por exemplo, pelo envio de uma mensagem à classe-pai.

Uma classe definida em Smalltalk herda automaticamente todos os elementos definidos na classe-pai e métodos podem ser sobrecarregados simplesmente pela definição com o mesmo nome de métodos da classe-pai. Quando o método sobrecarregado da classe-pai precisa ser usado, deve-se direcionar explicitamente o uso do método da classe superior com a palavra reservada `super` na chamada do método. Novas variáveis de instância também podem ser definidas nas subclasses.

Existe uma classe-raiz, a classe `Object`, e todas as demais classes são subclasses dessa. Toda nova classe tem uma única classe-pai (herança múltipla não é permitida) e posta na hierarquia de classes quando definida. Assim, todas as definições de classes, inclusive as predefinidas, estão dispostas em um ambiente único de hierarquia.

10.3.2 Variáveis e Vinculações

Variáveis em programas Smalltalk contêm sempre referências a objetos. E como a linguagem é projetada com vinculação dinâmica de tipo e métodos, todas as suas variáveis são polimórficas, isto é, podem fazer referência a objetos de quaisquer classes definidas no ambiente.

Com a vinculação dinâmica de tipos na linguagem, a vinculação dos métodos às mensagens também deve ser dinâmica. Uma mensagem enviada a um objeto faz com que seja pesquisada a existência do método na classe à qual o objeto pertence. Caso o método não exista naquela classe, a pesquisa procederá na classe-pai e assim por diante até que seja encontrada. Caso o método não seja encontrado em toda a cadeia ancestral, ocorrerá um erro. O método vinculado será sempre o mais próximo na hierarquia à classe do objeto (em tempo de execução).

O único erro relativo a tipos produzido na linguagem é quando o método não é encontrado na cadeia de classes ancestrais do objeto, dado que a tipagem da linguagem é dinâmica. Isso difere da maioria das linguagens e faz com que todo código em

Smalltalk seja genérico, no qual o significado de uma mensagem a um objeto só pode ser resolvido em tempo de execução porque as variáveis só possuem tipos em tempo de execução (polimorfismo dinâmico).

10.3.3 Expressões e Comandos

Os métodos em Smalltalk são basicamente definidos por expressões. As expressões definidas por literais e variáveis são na realidade objetos, como discutido anteriormente. As operações que podem ser realizadas sobre os literais e variáveis são o envio de uma mensagem aos respectivos objetos. Então, o envio de uma mensagem a um objeto é tido como uma expressão da linguagem, e o retorno da mensagem é o objeto valor da expressão.

Os métodos também são processados por uma expressão que retorna um objeto valor. Os comandos de atribuição são definidos de forma semelhante aos das linguagens imperativas, dado que as variáveis guardam referências aos objetos. A seleção é realizada por expressões definidas como métodos da classe `Boolean` (`ifTrue`, `ifFalse`). Os blocos de comando também são abstrações de funções que podem conter parâmetros de entrada e um parâmetro de retorno e são definidos como uma seqüência de expressões. Eles são instâncias da classe `Block`. Os métodos que definem as iterações estão também definidos nesta classe.

À parte da atribuição que é uma característica imperativa, as estruturas de controle da linguagem são realizadas pela passagem de mensagens para objetos. Isso faz com que toda a computação de programas Smalltalk possa ser vista como comunicação (passagem de mensagem) entre os objetos ativos, a idéia mais pura da orientação a objetos.

10.4 LEITURA RECOMENDADA

Uma discussão detalhada dos conceitos de orientação a objetos pode ser encontrada em [23] e [20]. Veja [20], em particular, para a discussão sobre a idéia de subclasses sob o ponto de vista de capacidade de substituição. Para uma visão mais completa

sobre tipos abstratos e sua relação com a programação orientada a objetos, consulte [21] e [24].

M. Abadi e L. Cardelli apresentam, em [1], uma teoria de objetos com uma definição acurada das noções de orientação a objetos. Esta leitura é recomendada aos interessados em semântica de linguagens de programação orientadas a objetos.

10.5 EXERCÍCIOS

1. A maioria das linguagens imperativas não possui classes e métodos. Como Você usaria uma linguagem imperativa como Pascal ou C para simular a idéia de classes e métodos?

2. A maioria das linguagens atuais não dá suporte à herança múltipla diretamente. Tente elaborar pelo menos dois problemas nos quais a herança múltipla é útil.

3. Ainda usando uma linguagem imperativa como C, como você simularia o mecanismo de herança? É possível simular herança múltipla nessa linguagem?

4. Ada possui formas de definir tipos abstratos. Discuta porque ela não é considerada uma linguagem orientada a objetos.

5. Argumente, sob o ponto de vista prático, porque as linguagens orientadas a objetos se tornaram tão populares e no que elas favorecem a reutilização.