Herança Simples, Agregação, Associação, Herança Múltipla

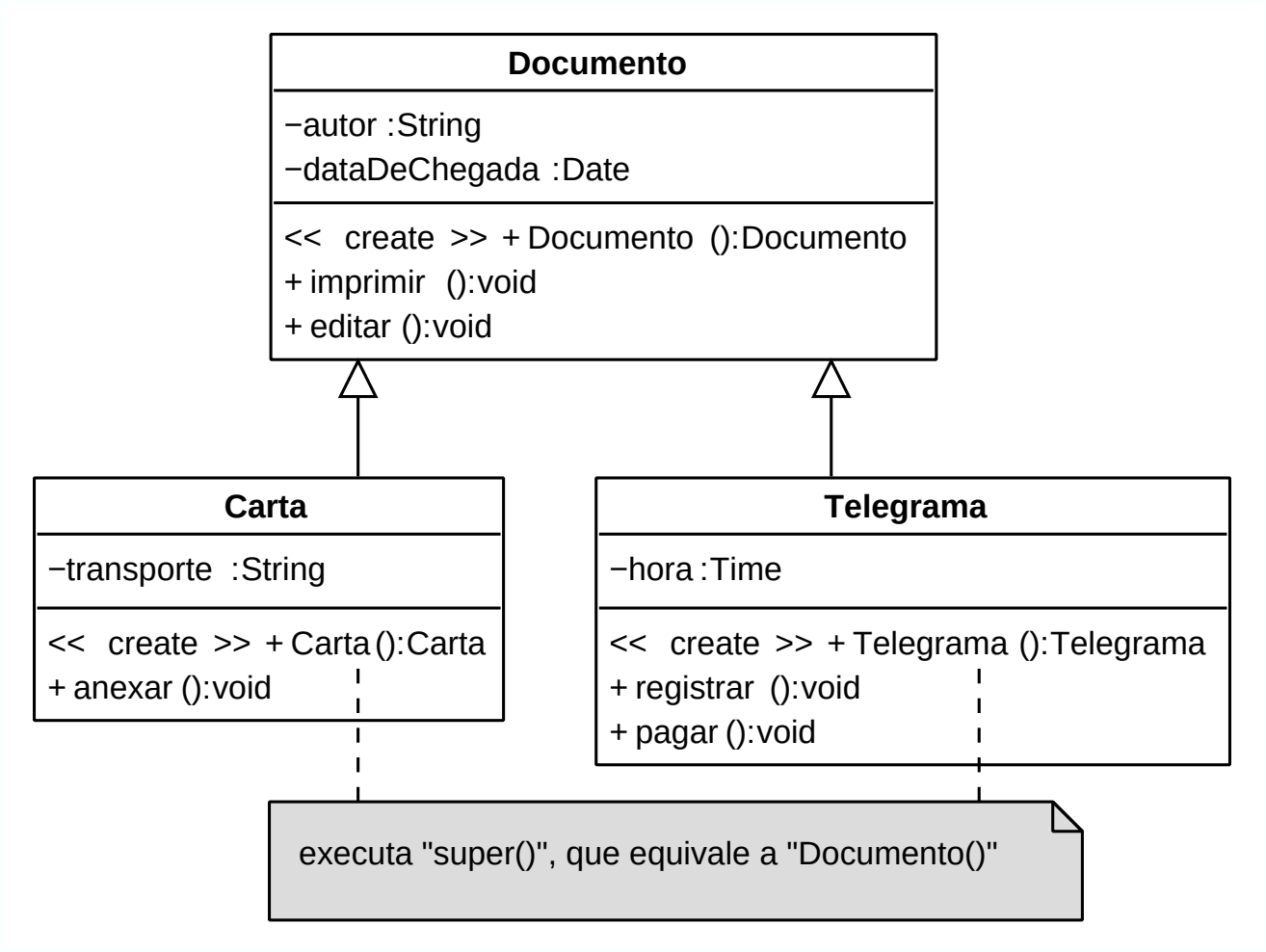
Herança Simples

- **Herança** (ou "subclassing") é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento
- Uma **classe derivada** herda a representação de dados e operações de sua **classe base**, mas pode:
  1. Adicionar novas operações e estender a representação de dados
  2. Redefinir a implementação de operações existentes
- Uma **superclasse** (ou classe base) proporciona a funcionalidade que é comum a todas as subclasses (ou classes derivadas), enquanto que uma **subclasse** proporciona a funcionalidade adicional que especializa o seu comportamento.

Métodos Construtores e Herança

- Os métodos construtores são exceções à regra de herança, pois não são herdados pelas subclasses.
- O construtor de uma classe tem que obrigatoriamente chamar um construtor de sua superclasse.
- Em Java, a palavra reservada **super** significa "minha superclasse" e o método **super()** significa "o método construtor da minha superclasse'.'
- O construtor pode receber parâmetros que são usados para inicializar atributos da superclasse.

Exemplo de Herança (I)



Exemplo de Herança (II)

```
class Documento {
private String autor;
private int dataDeChegada;

   public void criarDocumento(String s, int i){
      autor = s; dataDeChegada = i;
   }
   public void imprimir(){
      System.out.println(‘‘Imprime o Documento’’);
   }
   public void editar(){
      System.out.println(‘‘Edita o Documento’’);
   }
} // fim da classe Documento
```

Exemplo de Herança (III)

```
class Carta extends Documento {
 private String transporte;
 public Carta(){
   super(); //primeiro chama o construtor do pai
   transporte = "Aereo";
 }
 public void anexar(){
    ....
 }
} // fim da classe Carta
```

Definição de Tipos

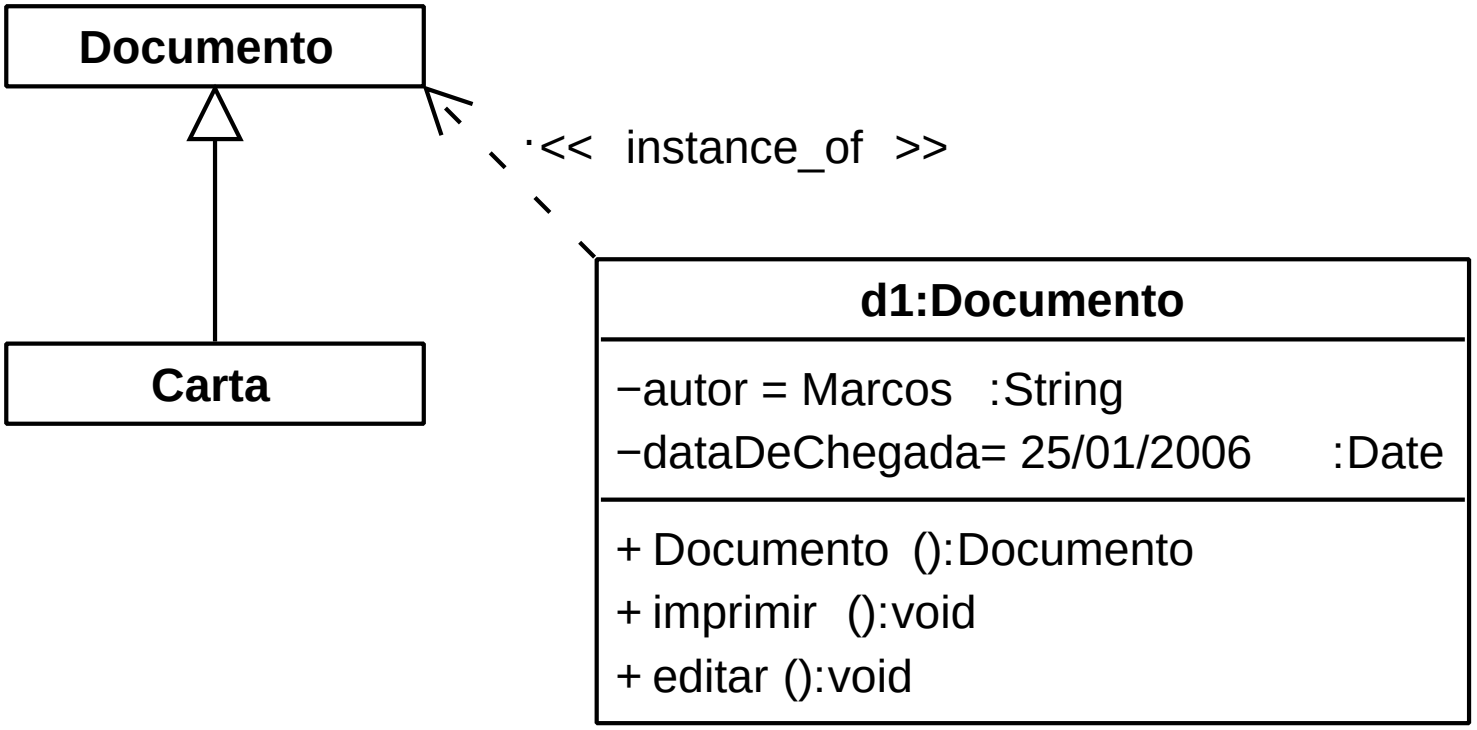
A hierarquia da classe `Documento` especifica 3 tipos diferentes:

```
TipoDocumento = {imprimir(), editar()}

TipoCarta = {imprimir(), editar(), anexar()}

TipoTelegrama = {imprimir(), editar(),
                 registrar(), pagar()}
```

OBS: Esses tipos estarão refletidos nas interfaces públicas dos objetos.

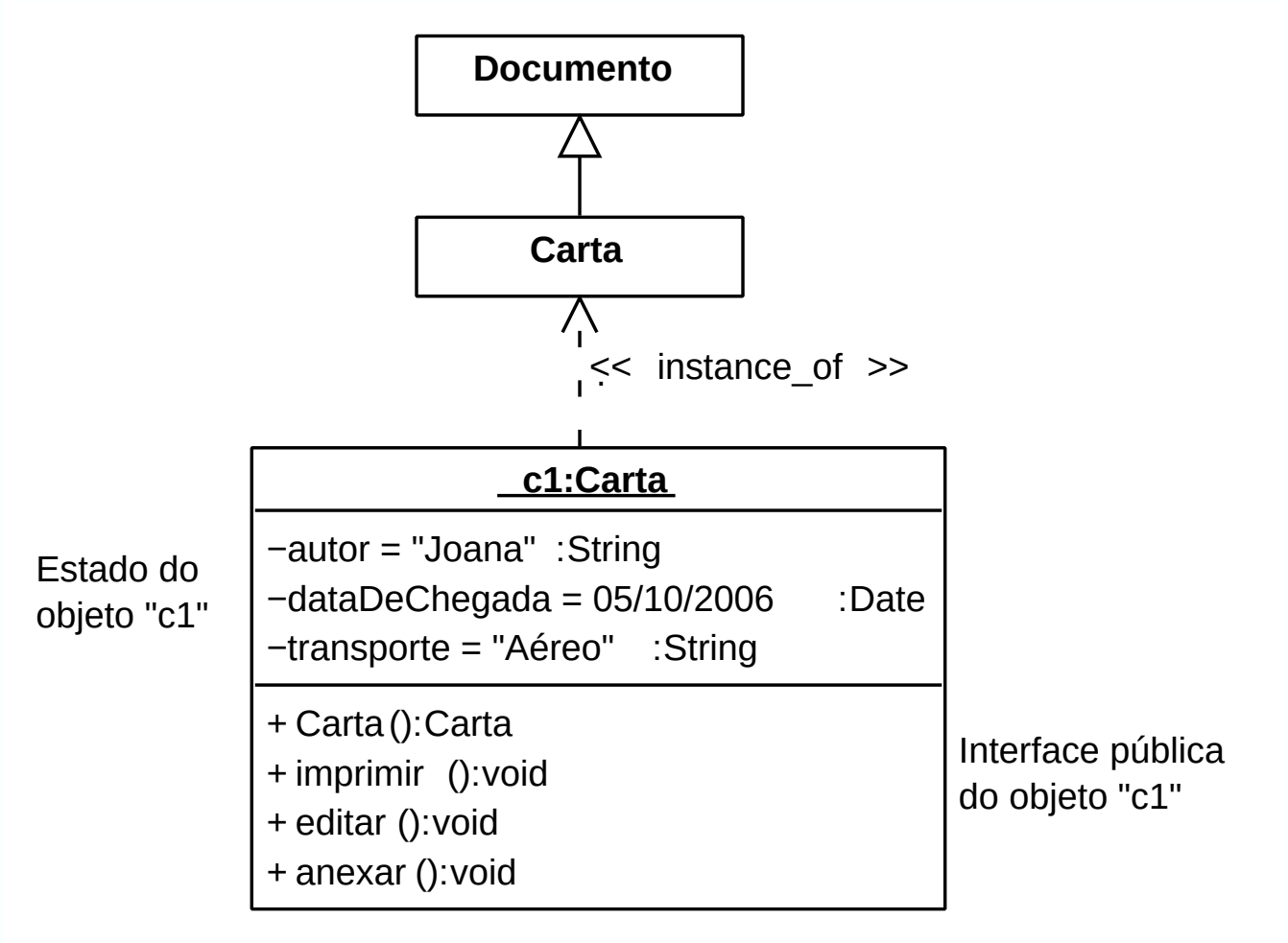
Criação de um objeto do tipo `Documento`

```
Documento d1 = new Documento();
```



Criação de um objeto do tipo Carta

```
Carta c1 = new Carta();
```



Visibilidade Privada (I)

O estado do objeto `c1` do tipo `Carta` contém 3 atributos:

`autor, dataDeChegada, transporte`

- Os atributos `autor` e `dataDeChegada` devem ser alterados apenas pelos métodos da classe `Documento` (visibilidade privada).

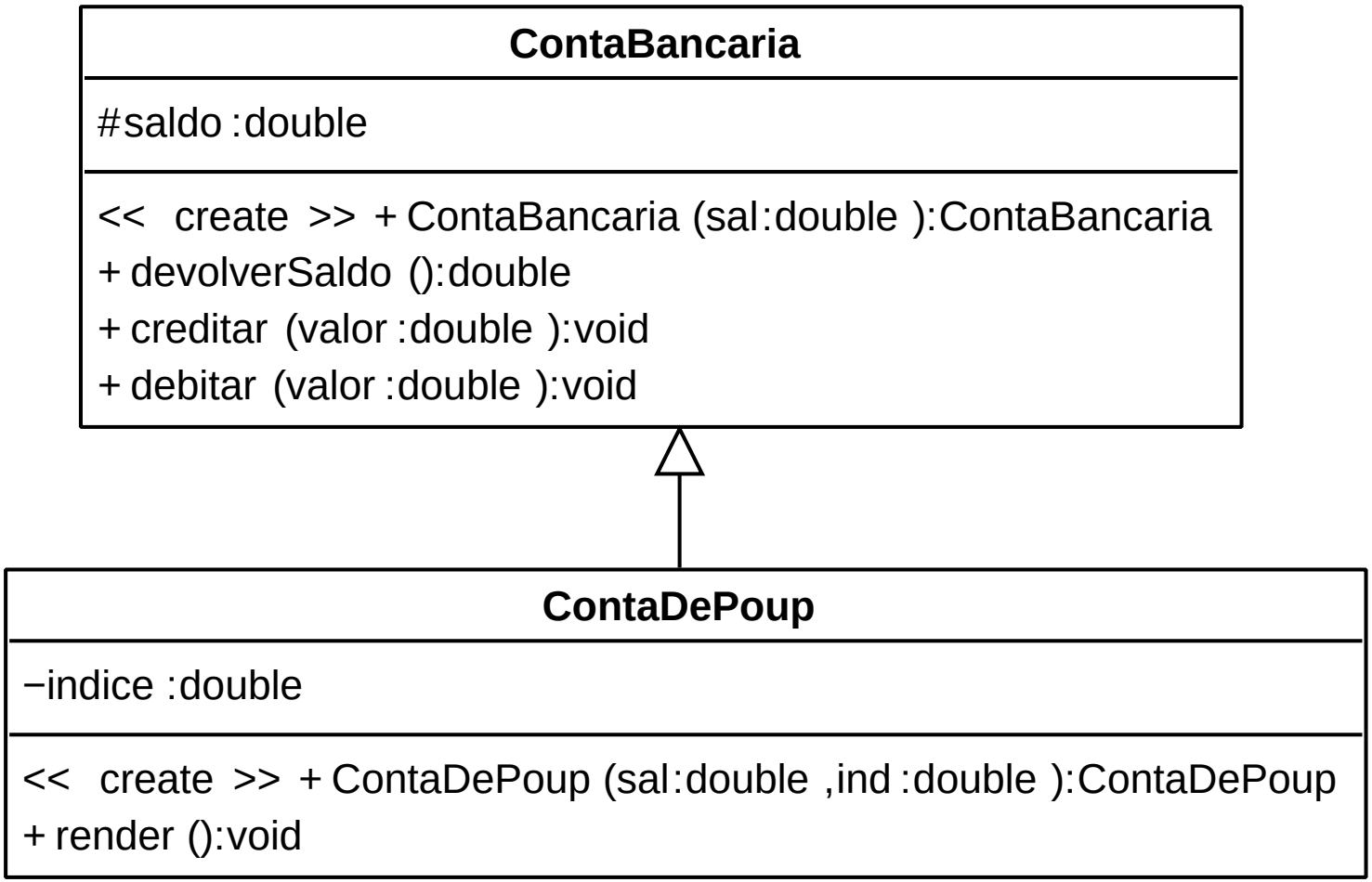
Visibilidade Privada (II)

- O atributo `transporte` deve ser alterado apenas pelos métodos da classe `Carta` (visibilidade privada).
- Os métodos da classe `Carta` não "enxergam" os atributos privados da superclasse e, portanto, ela não pode alterá-los diretamente.

Visibilidade Privada (III)

```
class Documento {
private String autor;
private int dataDeChegada;
public void editar(){// --implementação da operação
    autor = "Cecilia";
    dataDeChegada = 23032009;
 // erro: transporte = "Terrestre"; }
} // fim da classe Documento

class Carta extends Documento {
 private String transporte;
 public void anexar(){// --implementação da operação
    transporte = "Terrestre";
 // erro: autor = "Cecilia" dataDeChegada = 23/03/2009 }
} // fim da classe Carta
```

Exemplo de Herança



Visibilidade Protegida (I)

```
package A; //Obs: Em Java, superclasse deve ser definida
          // num pacote  diferente da subclasse
public class ContaBancaria {
 protected double saldo; // e se fosse private?
 public ContaBancaria(double sal){ //construtor não padrão
    saldo = sal;
 }
 public double devolverSaldo(){
    return saldo;
 }
 public void creditar(double valor){
    saldo += valor;}
 public void debitar(double valor){
    saldo -= valor;}
} // fim da classe ContaBancaria
```

Visibilidade Protegida (II)

```
package B; //subclasse definida no pacote B e
           // superclasse definida no pacote A

class ContaDePoup extends ContaBancaria{
 private double indice; // indice de rendimento
 public ContaDePoup(double sal, double ind){
     super(sal); // chamada p/ construtor de ContaBancaria
     indice = ind;
 }
 public void render(){//calcula e deposita o rendimento
    double i = indice * saldo; // ATENÇÃO!!
    saldo = saldo + i;         // ATENÇÃO!!
 }
} // fim da classe ContaDePoup
```

Visibilidade Protegida (III)

- A classe `ContaDePoup` altera um atributo declarado na classe base (saldo).
- Uma classe (mesmo sendo subclasse) não deveria alterar atributos declarados pelas suas superclasses: quebra de encapsulamento.
- O acesso é possível porque a visibilidade do atributo saldo é protegida na classe `ContaBancaria`.
- O que acontece se alterarmos a visibilidade protegida de saldo privada?

Visibilidade Protegida (IV)

- A implementação da operação render( ) na classe `ContaDePoup` deve ser mudada.

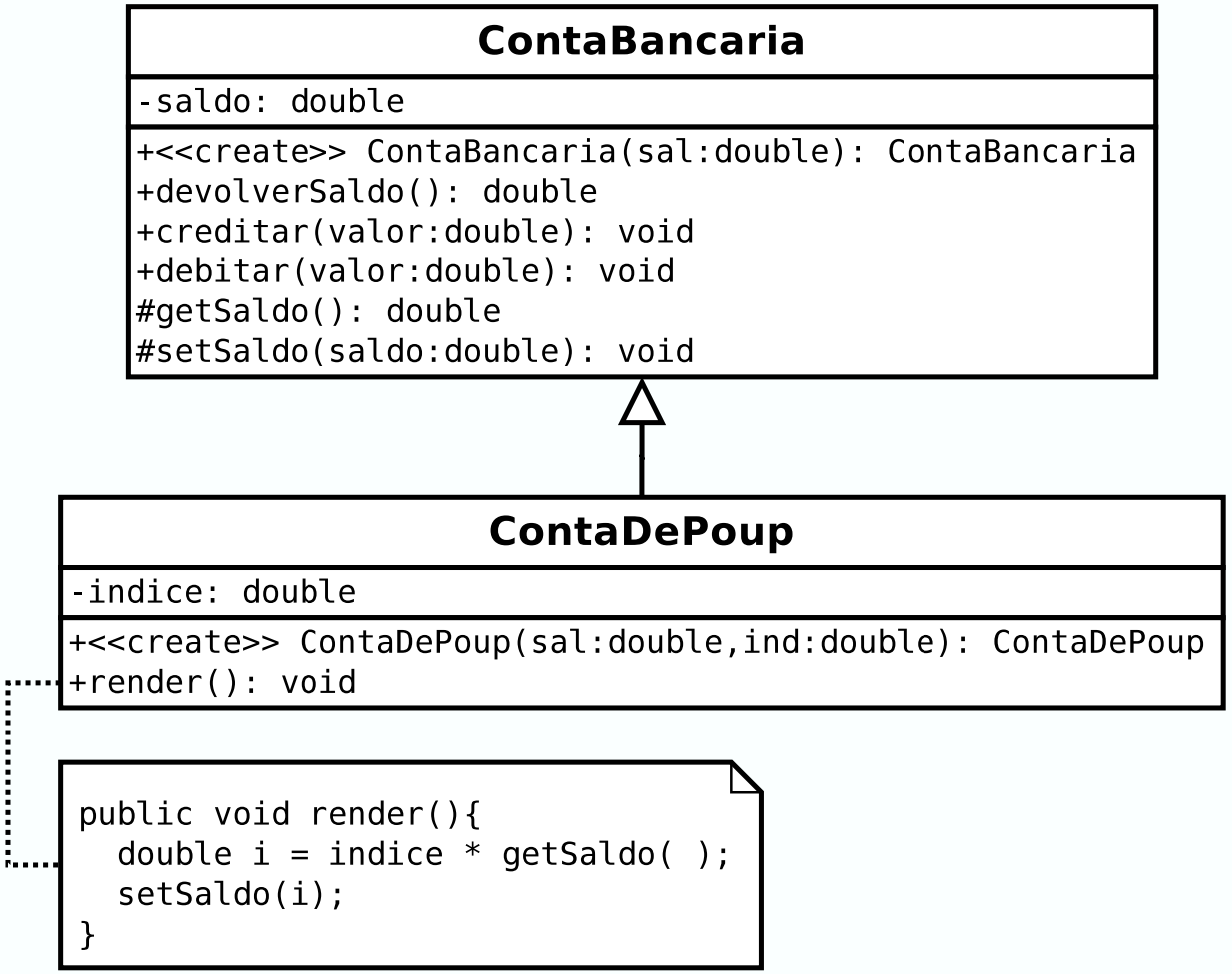
```
public void render ( ) {
   double s = this.devolverSaldo( );
   double i = indice * s;
   this.creditar(i);
}
```

- A classe `ContaDePoup` agora "não enxerga" o atributo `saldo`.
- O seu acesso deve ser realizado via a interface pública da classe `ContaBancaria`.
- A palavra reservada `this` pode ser usada dentro de um método para referenciar o objeto corrente que recebe a chamada do método.

Recomendações: Visibilidade Protegida (I)

- declarar atributos sempre privados.
- criar um conjunto de operações com visibilidade protegida que serão acessadas apenas pelas subclasses.
- lembrar que operações protegidas NÃO fazem parte da interface pública dos objetos e, portanto, não são visivéis externamente eles.

Recomendações: Visibilidade Protegida (II)



Recomendações: Visibilidade Protegida (III)

```
public class ContaBancaria {
 private double saldo; // sempre privado
 protected double getSaldo(){...} // não faz parte da
 protected setSaldo(double valor){...}//interface pública
 // interface pública da classe
 public ContaBancaria(double sal){...}
 public double devolverSaldo(){...}
 public void creditar(double valor){...}
 public void debitar(double valor){...}
} // fim da classe ContaBancaria
class ContaDePoup extends ContaBancaria{
....
public void render ( ) {
      double i = indice * getSaldo( ) ;
      setSaldo(i);} }// fim da classe ContaDePoup
```

Visibilidade Protegida: Java vs. C++

- O funcionamento da visibilidade protegida discutido no exemplo da hierarquia ContaBancaria vale para C++ sem a declaração de pacotes.
- Para Java, esse funcionamento é válido se a superclasse estiver num pacote diferente da sua subclasse.
- Se as duas classes estiverem no mesmo pacote, atributos e operações protegidos da superclasse podem ser acessados pelas subclasses e também por **TODAS** as outras classes não-derivadas pertencentes ao pacote.
- Ou seja, não existe visibilidade protegida entre superclasse e subclasse dentro do pacote. Isso se deve ao fato de Java implementar um quarto tipo de visibilidade (além de privada, pública e protegida), chamada de visibilidade de pacote.

Clientes por Herança X Clientes por Instanciação

Herança introduz dois tipos de clientes de uma classe:

- **Clientes por Instanciação:** os usuários criam instâncias da classe e manipulam-as usando suas interfaces públicas. Esses clientes são objetos.
- **Clientes por Herança:** os usuários são as próprias subclasses que herdam os métodos e atributos da superclasse. Esses clientes são classes.

Clientes por Instanciação

```
ContaBancaria c1 = new ContaBancaria();
double a = c1.saldo; //erro:visibilidade protegida
c1.render(); //erro: tipo ContaBancaria não define op
c1.getSaldo();//erro: msg não entendida pelo obj
c1.setSaldo(a);//erro: msg não entendida pelo obj

ContaDePoup c2 = new ContaDePoup();
double b = c2.saldo; // erro
double i = c2.indice; // erro
c2.render(); // válido
c2.creditar(10); // válido
c2.getSaldo();//erro: msg não entendida pelo obj
c2.setSaldo(a);//erro: msg não entendida pelo obj
```

Clientes por Herança

A classe `ContaDePoup` é cliente por herança e, portanto, tem acesso à parte pública e também à parte protegida, mas não tem acesso à parte privada da classe base.

As operações `getSaldo()` e `setSaldo()` podem ser chamadas apenas pela subclasse `ContaDePoup`, e não são visivéis nas interfaces dos objetos do tipo `ContaDePoup` ou `ContaBancaria`.

Os Modificadores de Visibilidade (I)

- **Pública:** qualquer categoria de cliente pode acessar, manipular e invocar diretamente atributos e métodos declarados como 'public'.
- **Privada:** nenhuma categoria de cliente pode acessar, manipular e invocar diretamente atributos e métodos declarados como "private". Somente a própria classe que os declara tem acesso direto.

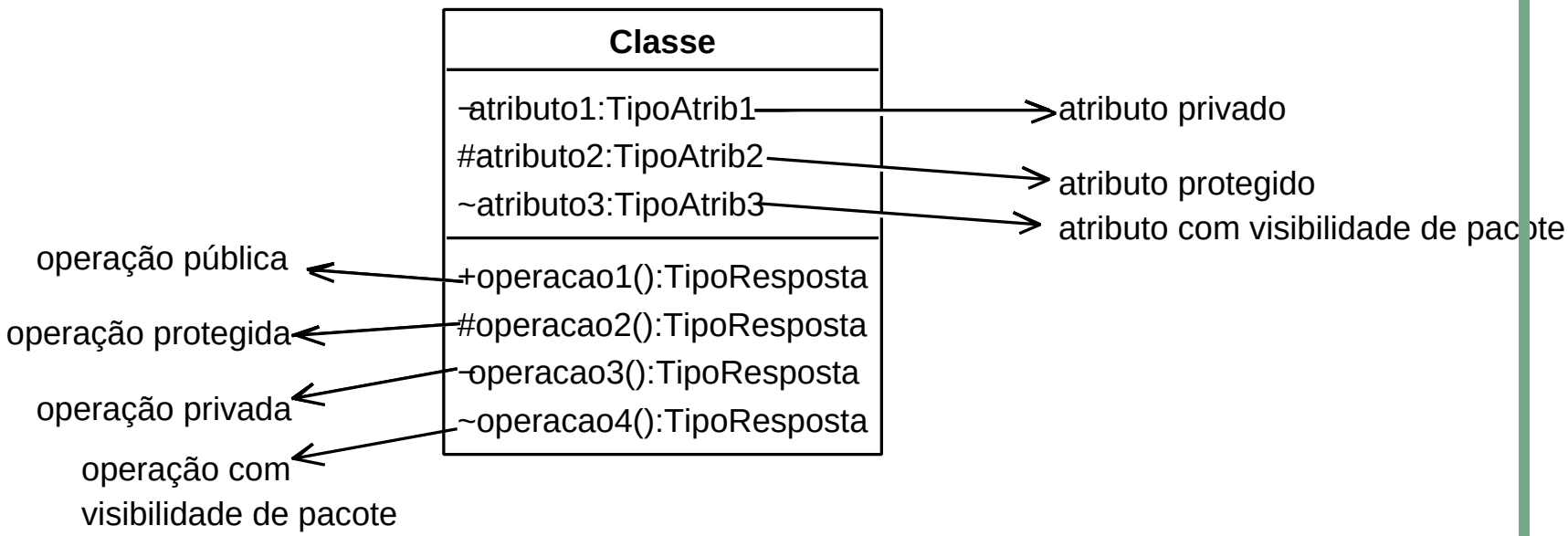
Os Modificadores de Visibilidade (II)

- **Visível pela subclasse (protegida):** se um atributo ou método é protegido, somente os clientes por herança têm acesso a ele. Os clientes por instanciação continuam a enxergar somente os atributos e métodos com visibilidade pública.
- **Visível no módulo (visibilidade de pacote):** se um atributo ou método possui visibilidade de pacote, somente os clientes que pertencerem ao mesmo módulo (pacote) podem acessá-lo diretamente. Os clientes externos continuam a enxergar somente os atributos e métodos com visibilidade pública.

Notação UML para Visibilidade

O modificador de visibilidade é representado antes do atributo ou operação. Existem 4 tipos diferentes de modificadores:

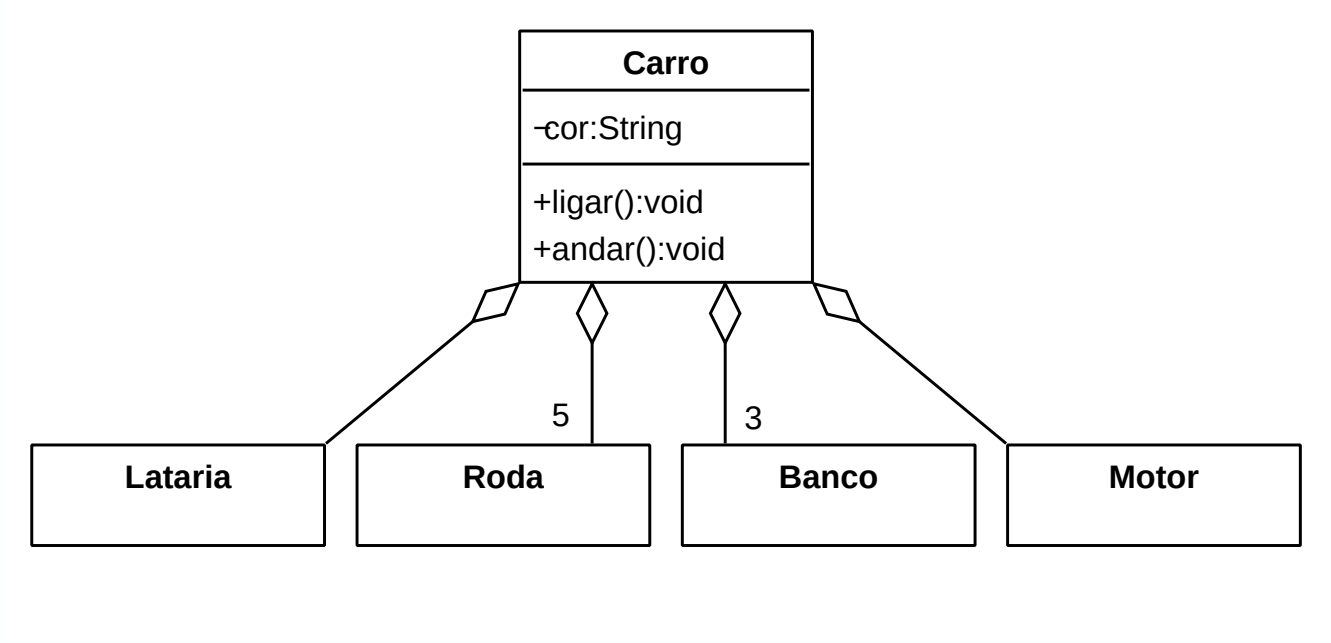
```
+ pública      - privada
# protegida    ~ visível dentro do pacote
```



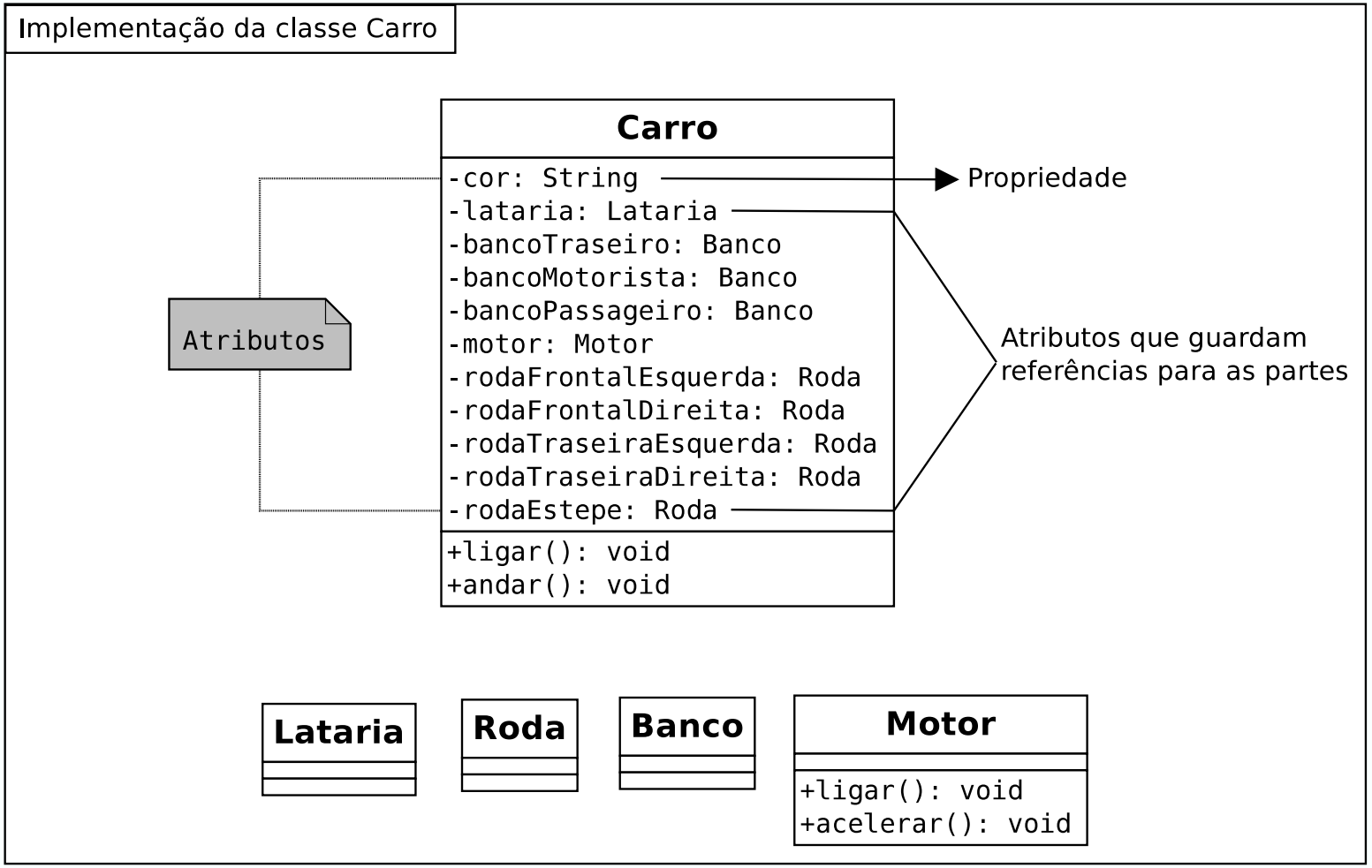
Implementação de Agregação (I)

- O relacionamento de agregação é materializado através de atributos;
- A classe agregadora (o todo) contém atributos que são referências (ponteiros) para os objetos agregados (que são as partes);
- A classe agregada (a parte) também pode conter um atributo que referencia o objeto agregador.

Implementação de Agregação (II)



Implementação de Agregação (III)



Implementação de Agregação (IV)

```
public class Carro {
   private String cor;
   private Lataria lataria;
   private Banco bancoTraseiro;
   private Banco bancoMotorista;
   private Banco bancoPassageiro;
   private Motor motor;
   private Roda rodaFrontalEsquerda;
   private Roda rodaFrontalDireita;
   private Roda rodaTraseiraEsquerda;
   private Roda rodaTraseiraDireita;
   private Roda rodaEstepe;
...
```

Implementação de Agregação (V)

```
   public Carro(String cor) {
      this.cor = cor;
      lataria = new Lataria();
      bancoTraseiro = new Banco();
      bancoMotorista = new Banco();
      bancoPassageiro = new Banco();
      motor = new Motor();
      rodaFrontalEsquerda = new Roda();
      rodaFrontalDireita = new Roda();
      rodaTraseiraEsquerda = new Roda();
      rodaTraseiraDireita = new Roda();
      rodaEstepe = new Roda();
   }//fim do construtor
...
```

Implementação de Agregação (VI)

```
   public void ligar() {
      motor.ligar(); //propagação de operações
                     //  trigerring (disparo)
   }//fim do método ligar

   public void andar() {
      motor.acelerar(); //propagação de operações
   }//fim do método andar

}//fim da classe Carro
```

Agregação

- Pode ser lida nas duas direções (bidirecional): um carro (objeto agregador) é feito de componentes (partes).
- Os componentes fazem parte do carro.
- A agregação é transitiva: o todo possui partes, que por sua vez, podem ter partes.
- A agregação recursiva é comum.
- A propagação de operações é um bom indicativo da existência da agregação.

Agregação – Exercício

- O código da hierarquia de Carro mostrada anteriormente implementa a agregação do todo com suas partes de forma unidirecional.
- Modifique o código para implementar a agregação biderecional (i.e. as partes conhecendo o seu todo).

Implementação de Agregação vs. Associação

- A associação descreve um conjunto de conexões com estrutura e semântica comuns: uma pessoa trabalha para uma empresa.
- A agregação é uma forma especial de associação que acrescenta conotações semânticas extras.
- A associação é implementada da mesma forma que a agregação, i.e., através de um atributo de um objeto que contém uma referência explícita para outro objeto.

Usos de Herança

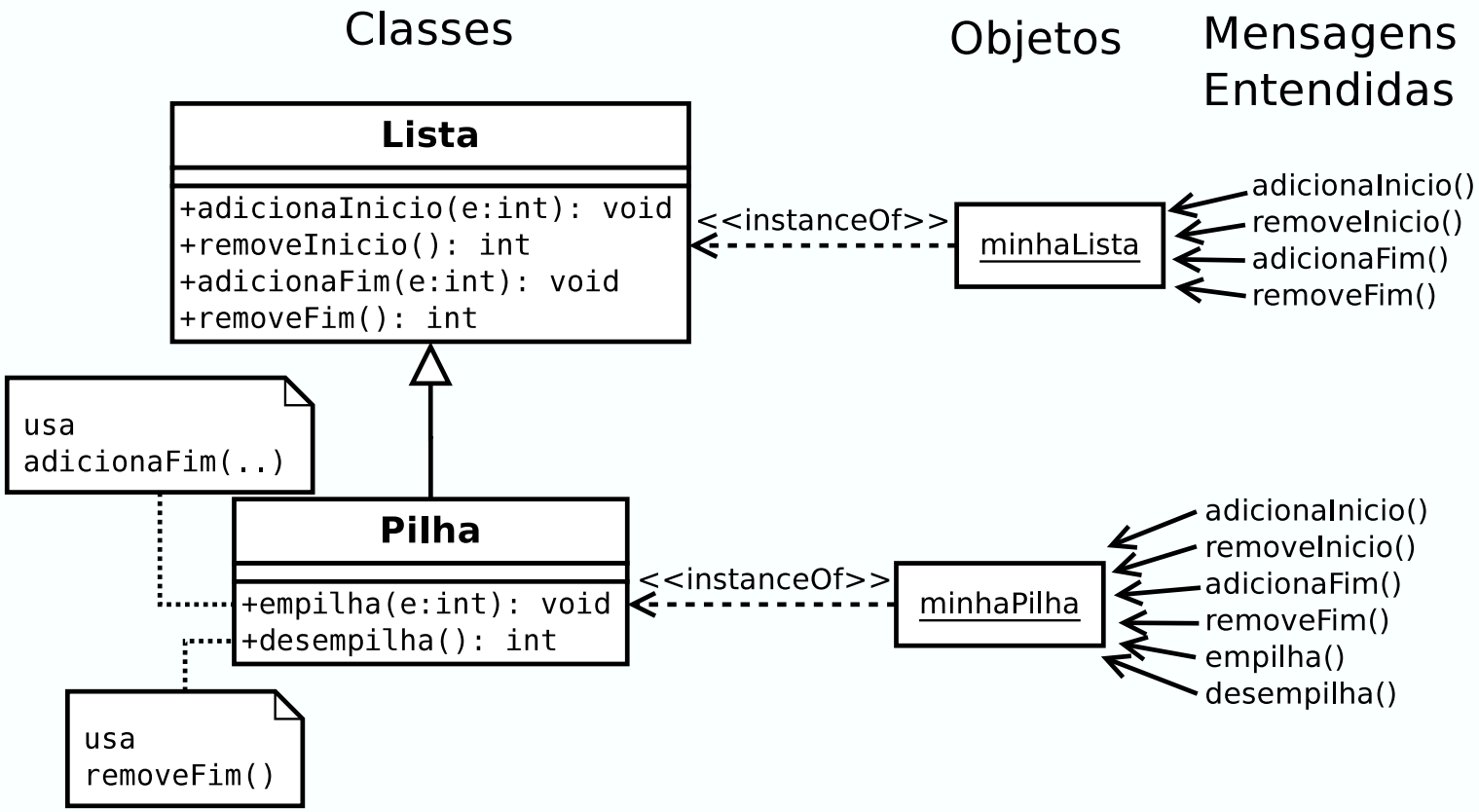
O mecanismo de herança permite a construção de duas categorias de hierarquias:

- Hierarquias de implementação
- Hierarquias de comportamento

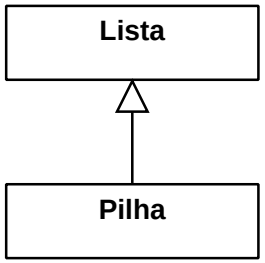
Hierarquias de Implementação

- Herança é usada como uma técnica para implementar TADs que são similares a outros já existentes (reutilização de código)
- Nesse caso, o programador usa herança como uma técnica de implementação, com nenhuma intenção de garantir que a subclasse tenha o mesmo comportamento da superclasse
- Você pode herdar comportamento não desejado, implicando num comportamento INCORRETO da subclasse

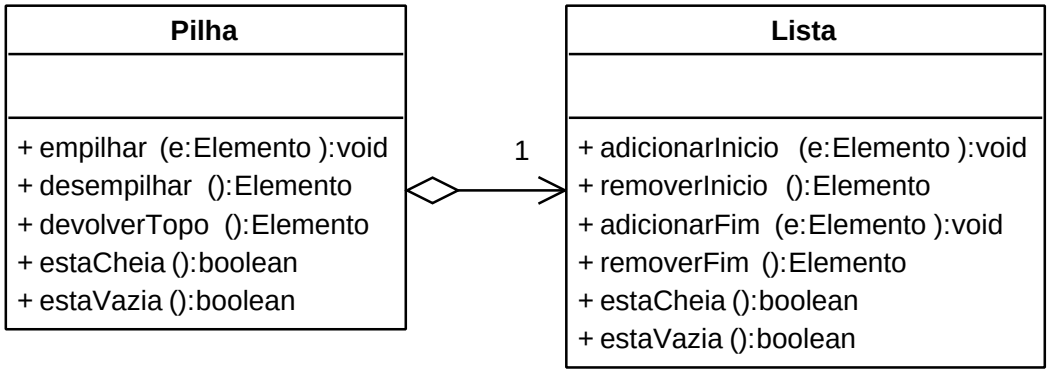
Exemplo: Pilha



Solução Recomendada em Java (I)

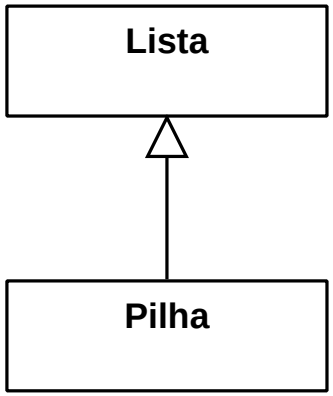


NÃO RECOMENDADO

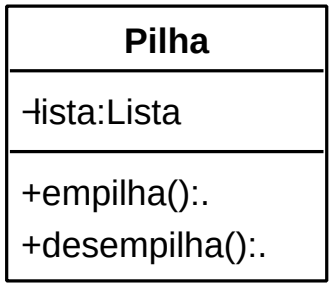


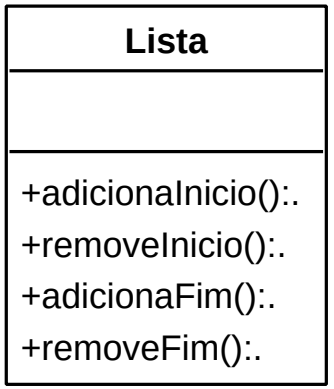
RECOMENDADO

Solução Recomendada em Java (II)



NÃO RECOMENDADO





RECOMENDADO

Solução Usando Agregação (Java)

```
class Pilha{
  private Lista lista;
       // apontador para um objeto da Lista
       // este atributo implementa o
       // relacionamento de agregação da hierarquia
  public Pilha(){lista = new Lista()}
  public void empilha(int x){lista.adicionaFim(x);}
  public int desempilha(){return lista.removeFim();}
}
```

Derivação Pública

- **Java:***class Derivada extends Base...*
- **C++:** *class Derivada: public Base ...;*
- Todos os membros **públicos** da classe Base tornam-se membros **públicos** da classe Derivada.

Exemplo de Derivação Pública em C++ (I)

```
class Lista{ // lista de inteiros
   public:
      Lista(){...}
      void adicionaInicio(int x){
            // x é a nova cabeça da lista
      }
      int removeInicio(){
          // remove e retorna o valor final da lista
          // pré-condição: !estaVazia()
      }
```

Exemplo de Derivação Pública em C++ (II)

```
        int removeFim(){
              // remove e retorna o último
              // elemento da lista
              //pré-condição: !estaVazia()
        }
        void adicionaFim(int x){
              // x torna-se o último elemento da lista
        }
        ...
}
```

Exemplo de Derivação Pública em C++ (III)

```
class Pilha : public Lista{
   public:
      Pilha(){...}
      void empilha(int x){adicionaFim(x);}
      int desempilha(){return removeFim();}
}
     ...
Pilha* plh = new Pilha();
plh->adicionaInicio(37); // corrompe o estado da pilha
```

- Note que a interface de `plh` também herda operações que não têm nada a ver com a semântica de pilhas (herança de implementação).

Derivação Privada (C++) (I)

- *class Derivada: private Base ...;*
- Todos os membros **públicos** da classe Base tornam-se membros **privados** da classe Derivada.

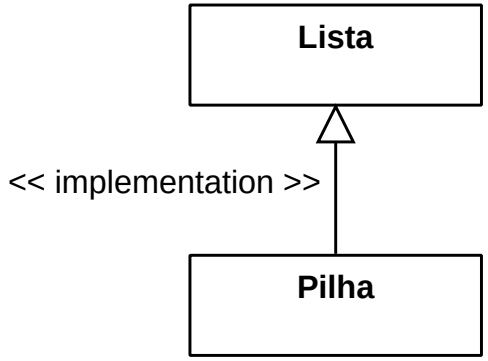
```
class Pilha : private Lista{
  public:
    Pilha(){}
    void empilha(int x){adicionaFim(x);}
    int desempilha(){return removeFim();}
};
Pilha* plh = new Pilha();
plh.adicionaInicio(37); // erro de compilacao
```

- **Por quê?** Restrição de visibilidade (herança X instanciação)

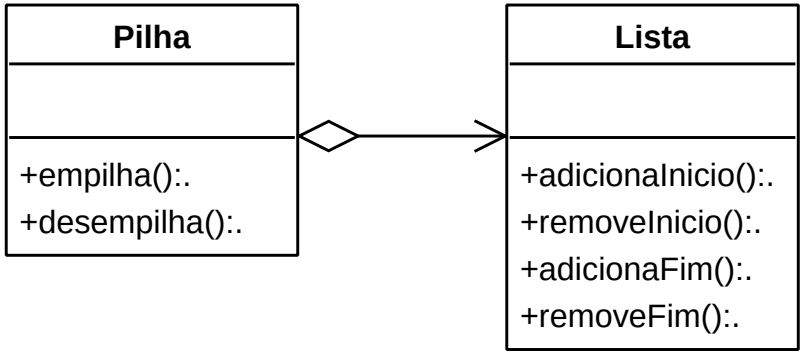
<<Implementation>> em UML(I)

- Em UML, o estereótipo <<*implementation*>> aplicado ao relacionamento de generalização especifica que a classe derivada herda as implementações da superclasse mas não faz com que suas operações sejam públicas na interface da subclasse (derivação privada de C++).

<<Implementation>> em UML(II)



NÃO RECOMENDADO

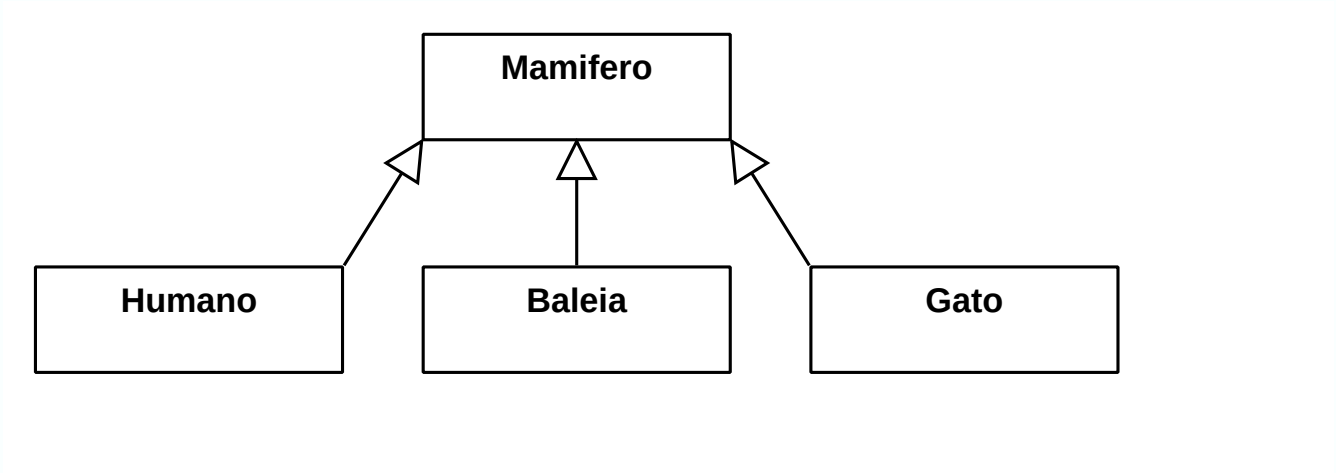


RECOMENDADO

Solução Usando Agregação (C++)

```
class Pilha{
   private:
      Lista* lista; // apontador para
                    //um objeto do tipo Lista
   public:
      Pilha(){lista = new Lista();}
      void empilha(int x){lista->adicionaFim(x);}
      int desempilha() {return lista->removeFim();}
      Boolean estaVazia(){return lista->estaVazia();}
      Boolean estaCheia(){return lista->estaCheia();}
}
```

Herança de Comportamento (I)



Herança de Comportamento (II)

- Herança de comportamento representa uma hierarquia verdadeira de generalização/especialização
- Herança de comportamento equivale ao relacionamento **é-um**, ou **é-subtipo-de**
- No exemplo, podemos dizer que o tipo `Humano` é um tipo especializado de `Mamífero` ou que o tipo `Humano` é um subtipo de `Mamífero`

O Conceito de Subtipo (I)

- Subtipo preocupa-se com o **compartilhamento de comportamento** ("behaviour sharing")
- Então, se S é um subtipo de T, um objeto do tipo S pode ser usado no lugar de um objeto de tipo T
- A idéia principal é que classes derivadas "comportem-se como se fossem as classes bases"

O Conceito de Subtipo (II)

```
Humano h = new Humano();
Baleia b = new Baleia();
Gato g = new Gato();
Mamifero m;
m = h; // princípio da substitutabilidade
m.mamar(); //ok
m.falar(); // erro
m = b; // princípio da substitutabilidade
m.mamar(); //ok
m.respirarPeloEspiraculo(); // erro
m = g; // princípio da substitutabilidade
m.mamar(); //ok
m.miar(); // erro
```

O Conceito de Subtipo (III)

- Dizemos que o subtipo S se "conforma" ao tipo "T", i.e., "S está de acordo com T", "S está conforme com T".
- No modelo de objetos, o compartilhamento de comportamento é somente justificável quando existe um relacionamento verdadeiro de generalização/especialização entre as classes.
- No caso, hierarquia de generalização/especialização definida por Lista e Pilha não é verdadeira, pois existem operações de Lista que invalidam o comportamento da classe Pilha.
- Portanto, você não deveria usar herança, e sim o relacionamento de agregação.

Exemplo de Uso de Herança (I)

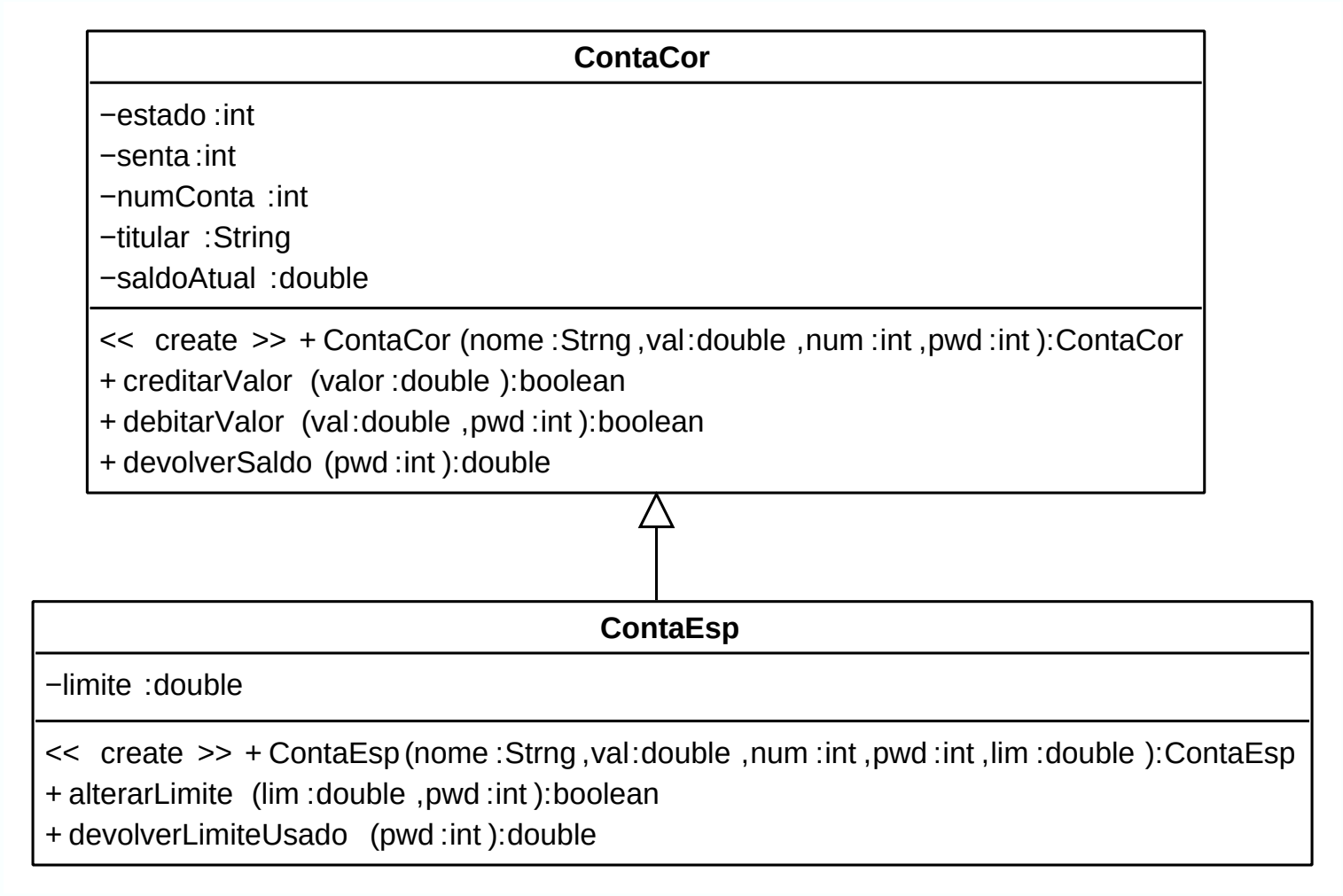
Descrição do Problema

Em um sistema bancário, uma classe **ContaCor**, que representa uma conta corrente, deve oferecer três operações: (i) creditarValor(), (ii) debitarValor() e (iii) devolverSaldo().

Além disso, uma conta corrente pode ser classificada como especial (**ContaEsp**), que é um tipo de conta corrente. Diferentemente de uma conta corrente comum, ao se criar uma conta corrente especial, é adicionado um valor de R$ 200,00 ao saldo da conta. Esse valor se refere ao limite de crédito oferecido pelo banco.

A classe **ContaEsp** define duas novas operações: (i) alterarLimite() e (ii) devolverLimiteUsado().

Exemplo de Uso de Herança (II)



Classe ContaCor

```
// arquivo ContaCor.java
class ContaCor {
   private int estado; // 1 = ativo ; 2 = inativo
   private int senha;
   private int numConta;
   private String titular;
   private double saldoAtual;
   public ContaCor(String nome, double val,
                   int num ,int pwd) {...}  }
   public boolean creditarValor(double val) {...}
   public boolean debitarValor(double val, int pwd) {...}
   public double devolverSaldo(int pwd) {...}
}
```

Classe ContaEsp (I)

```
//arquivo ContaEsp.java
class ContaEsp extends ContaCor{
   private float limite;
   ContaEsp(String nome, float val, int num,
            int pwd, double lim){
      super(nome, val, num, pwd); // construtor da
                                  // superclasse
      limite = lim;
      creditarValor(limite);
   } // fim do construtor ContaEsp()
```

Classe ContaEsp (II)

```
public boolean alterarLimite(double lim, int pwd){
   boolean r;
   if(lim > limite) // aumenta o limite
     r = this.creditarValor(lim-limite);
   else // redução de limite
     r = this.debitarValor(limite-lim, pwd);
   if(r)
     limite = lim; // se lançamento ok,
                   //altera o limite atual
   return r;
} // fim de alterarLimite
```

Classe ContaEsp (III)

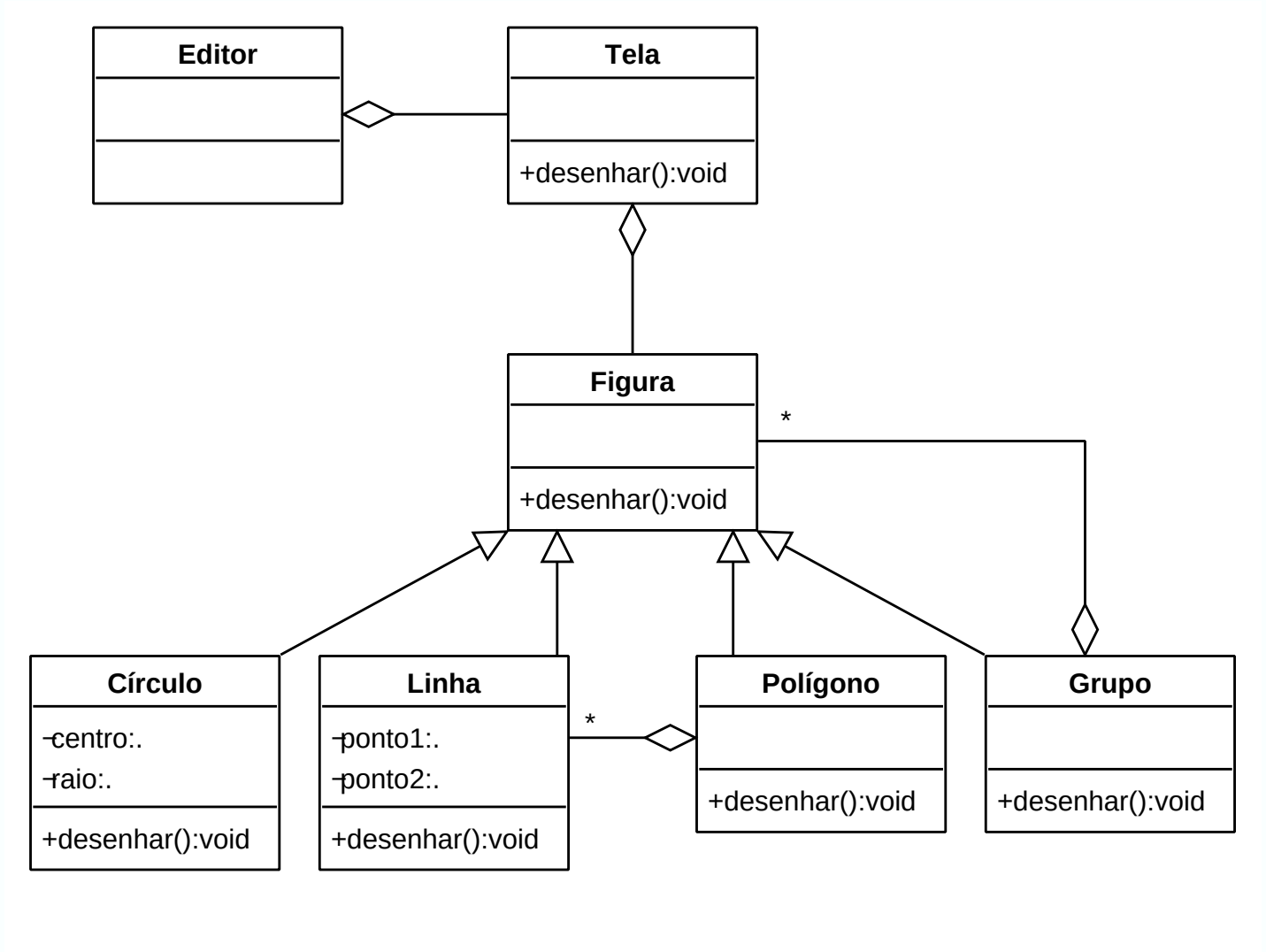
```
    public double devolverLimiteUsado(int pwd){
       double s;
       s = this.devolverSaldo(pwd); // obtem saldo atual
       if (s == -1) // senha invalida
          return -1; // fracasso
       if(s > limite)
          return 0;
       else
          return (limite - s);
    } // fim de devolverLimiteUsado()
}
```

Exercício 1

Um editor de desenhos geométricos possui uma tela para desenhar. Essa tela pode ser constituída de muitas figuras. Figuras podem ser círculos, linhas, polígonos e grupos. Um grupo consiste de muitas figuras e um polígono é composto por um conjunto de linhas. Quando um cliente pede para que uma tela se desenhe, a tela, por sua vez, pede para cada uma das figuras associadas que se desenhe. Da mesma forma, um grupo pede que todos os seus componentes se desenhem. Crie uma hierarquia de generalização/especialização que classifique essas diferentes figuras geométricas e identifique o comportamento de cada tipo abstrato de dados que você criar, bem como os seus respectivos atributos.

Resposta – Exercício 1

Resposta – Exercício 1



Exercício 2

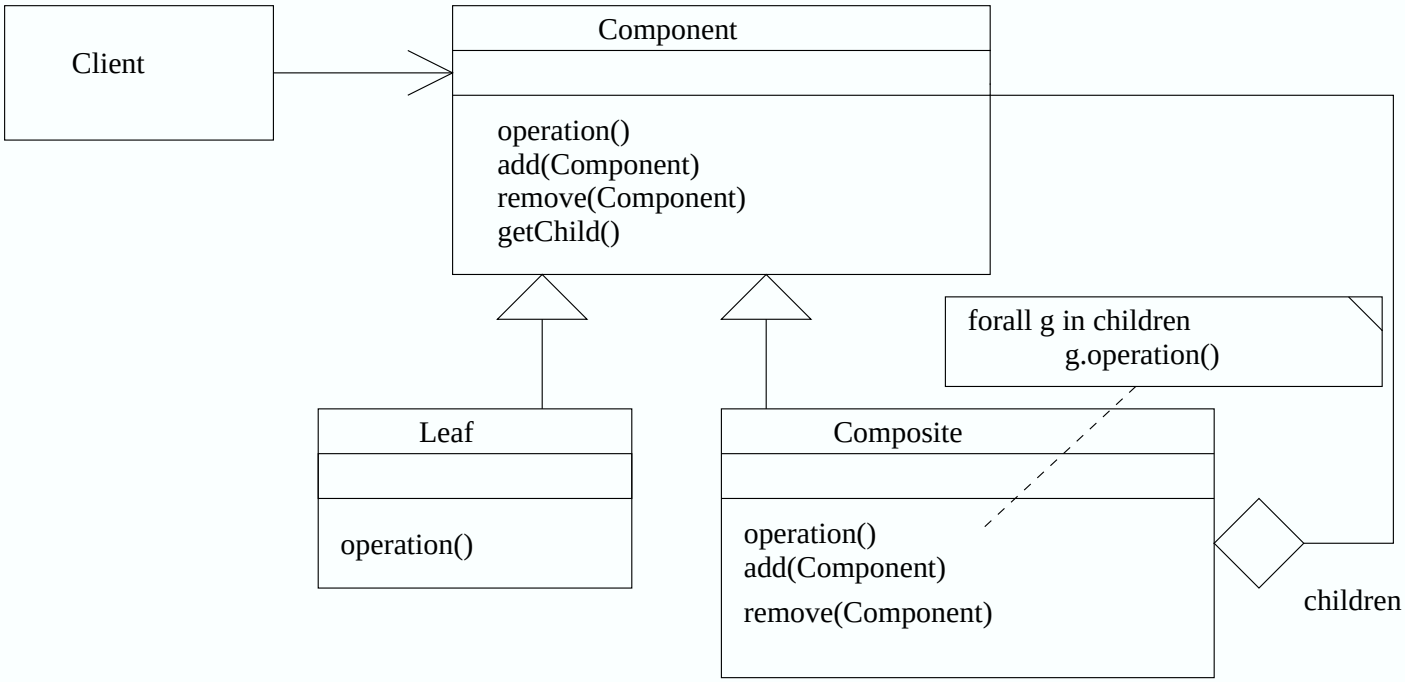
Uma expressão aritmética consiste de um operando, um operador (+ - * /) e um outro operando. Um operando pode ser um número ou uma outra expressão aritmética. Portanto, 2 + 3 e (2 + 3) + (4 * 6) são ambas expressões válidas. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Design Pattern Composite

Gang of Four (GoF) Patterns. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. First edition, Addison-Wesley, 1995.

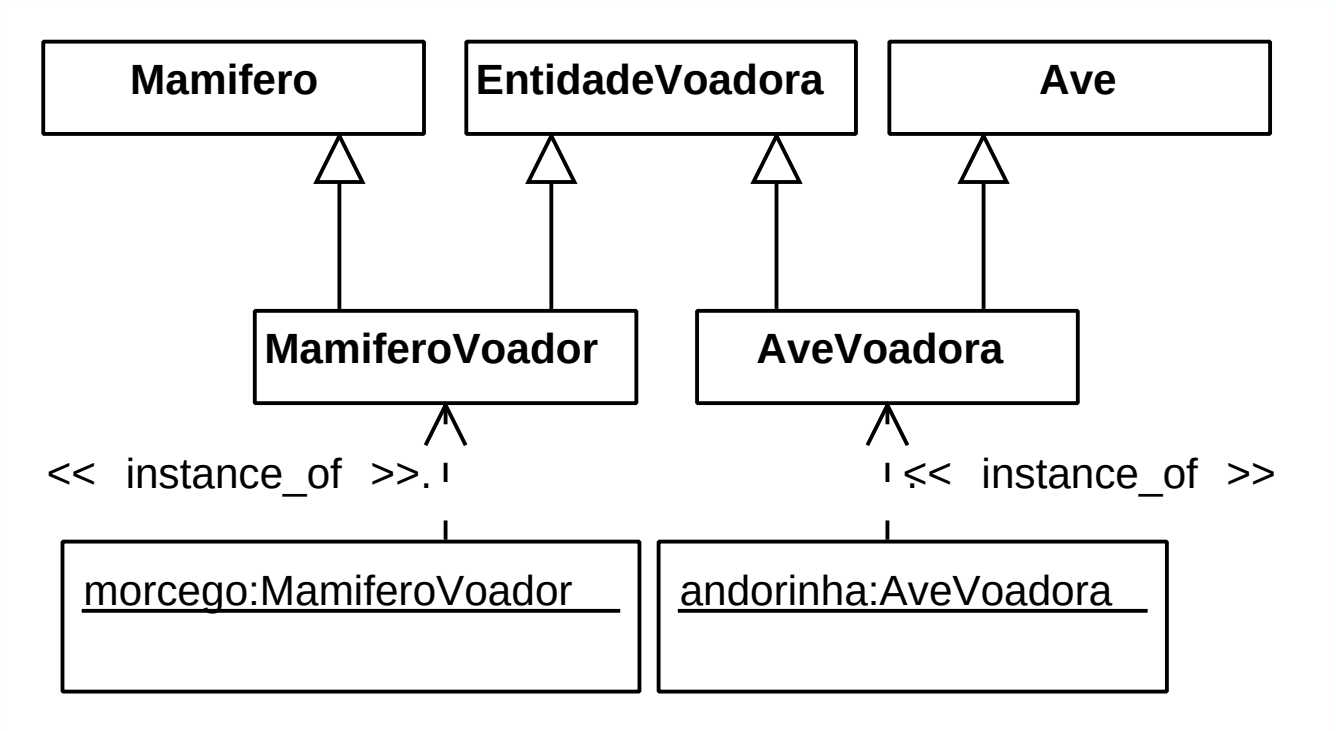
http://www.vincehuston.org/dp/

http://pt.wikipedia.org/wiki/Composite

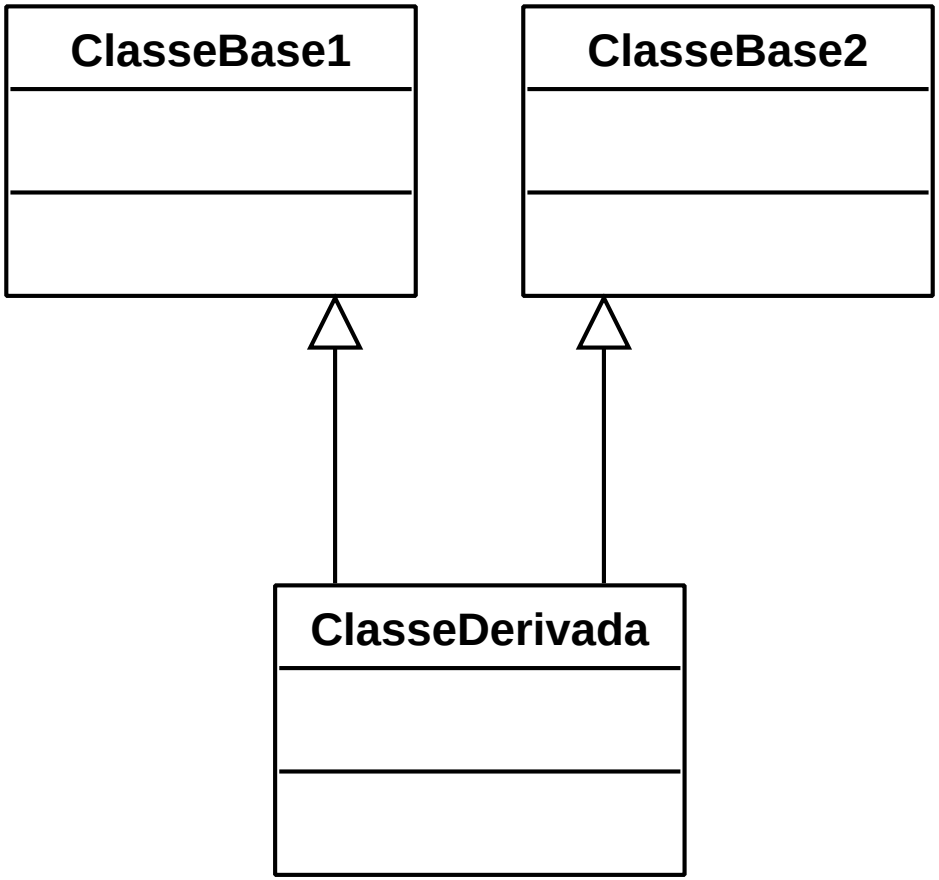


Herança Múltipla de Classes

Herança Múltipla de Classes (I)



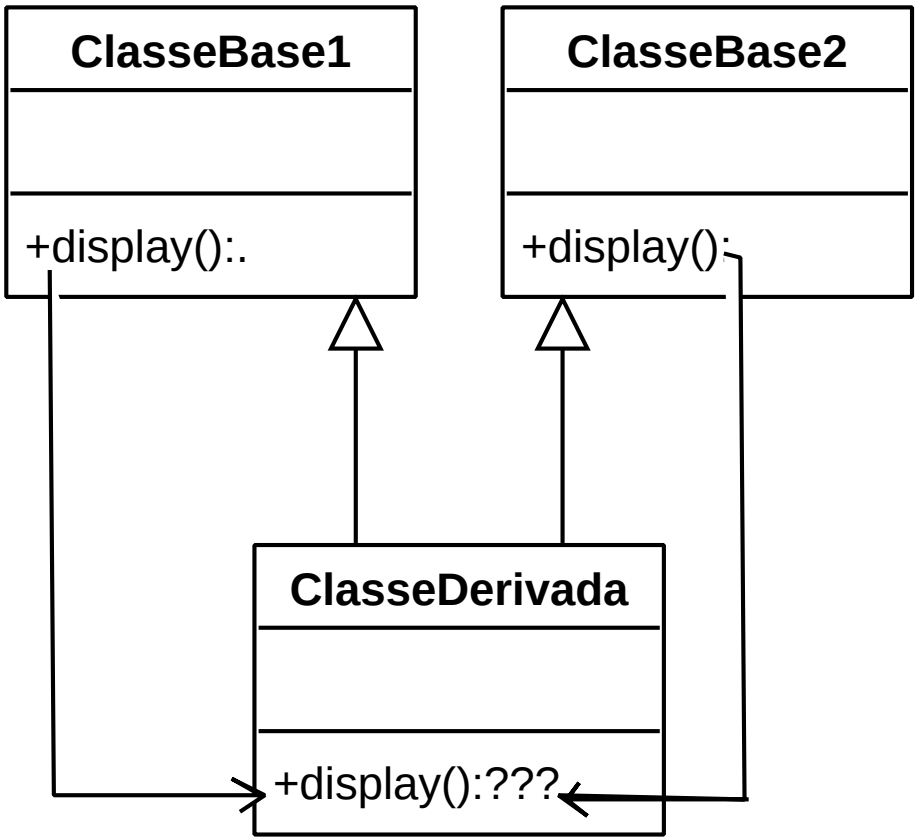
Herança Múltipla de Classes (II)



Problemas com Herança Múltipla (I)

Quando uma classe herda de mais de um pai, existe a possibilidade de conflitos, já que operações e atributos com mesmo nome e semânticas diferentes podem ser herdadas de superclasses diferentes

Problemas com Herança Múltipla (II)



Estratégias para a Resolução de Conflitos

- **Linearização:** especifica uma ordem linear das classes (Flavors e CommonLoops)
- **Renomeação:** alteração dos nomes de atributos e operações com conflitos (Eiffel)
- **Operador de Qualificação:** sempre que ocorrer ambigüidade deve-se usar "::" (C++)

Conflito de Nomes em Eiffel (I)

- Em caso de conflito de nomes, a regra em Eiffel é muito simples, tais conflitos são proibidos
- Assim, se as classes $ClasseBase1$ e $ClasseBase2$ têm uma mesma operação $display()$ e é definida a classe $ClasseDerivada$ como:

```
class ClasseDerivada export...inherit
   ClasseBase1;
   ClasseBase2
feature ... end
```

esta classe é rejeitada pelo compilador

Conflito de Nomes em Eiffel (II)

- O conflito de nomes pode ser removido introduzindo 1 ou mais subcláusulas **rename** na cláusula **inherit** da subclasse.

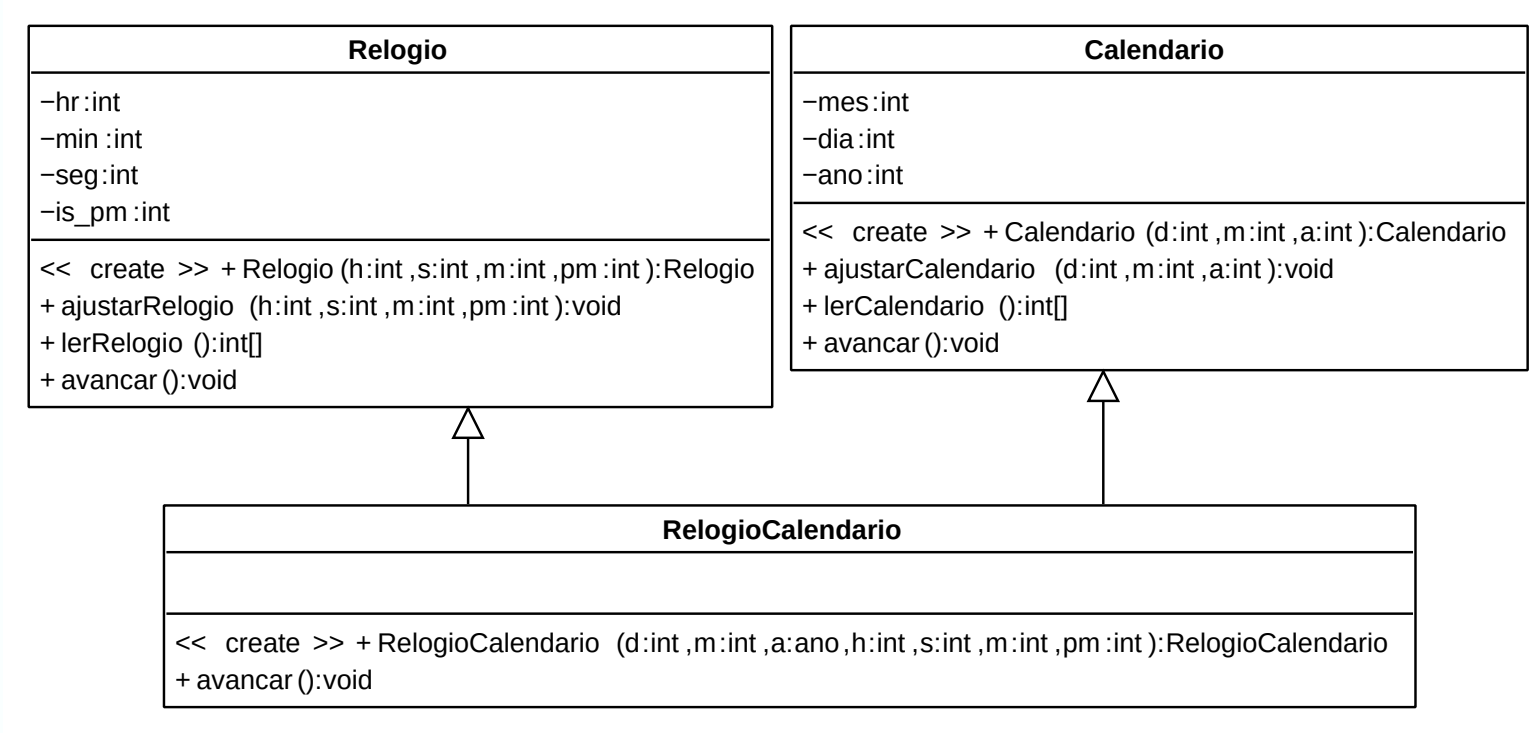
```
class ClasseDerivada export...inherit
  ClasseBase1 rename display() as ClasseBase1_display();
  ClasseBase2
feature ... end
```

- Dentro da *ClasseDerivada*, a operação *display*() da *ClasseBase*1 será chamada como *ClasseBase*1_*display*() e a da *ClasseBase*2 será chamada como *display*().
- Clientes por instanciação da ClasseBase1 usam a operação *display*(), e não a renomeada.

Conflito de Nomes em Eiffel (III)

```
ClasseBase1 cb1;
ClasseBase2 cb2;
ClasseDerivada cd = new ClasseDerivada();

cb2 = cd; // é válido
cb2.display(); // usa ClasseBase2::display()

cb1 = cd; // é válido
cb1.display(); // usa ClasseBase2::display()
cb1.ClasseBase1_display(); //ERRO

cb1 = new ClasseBase1();
cb1.display(); // usa ClasseBase1::display()
```

Herança Múltipla em C++ (I)



Herança Múltipla em C++ (II)

```
class Relogio {
  protected:
    int hr;  int min;
    int seg; int is_pm;
  public:
    Relogio( int h, int s, int m, int pm){...}
    void ajustarHora(int h, int s, int m, int pm){...}
    void lerRelogio(int& h, int& s, int& m, int& pm){...}
    void avancar(){...} // ------------> conflito de nome
};
```

Herança Múltipla em C++ (III)

```
class Calendario {
  protected:
    int mes;
    int dia; int ano;
  public:
    Calendario( int d, int m, int a){...}
    void ajustarCalendario(int d, int m, int a){...}
    void lerCalendario(int& d, int& m, int& y){...}
    void avancar(){...} // ----------> conflito de nome
};
```

Herança Múltipla em C++ (IV)

```
class RelogioCalendario : public Relogio,
                          public Calendario {
  public:
    RelogioCalendario (int d, int m, int a, int h,
                       int mn, int s, int pm){...}
    void avancar(){
            Relogio::avancar();
            Calendario::avancar(); }
            //---------> Métodos Qualificados
};
```

Obs: Em C++, é obrigatório a redefinição da operação com conflito na classe derivada.

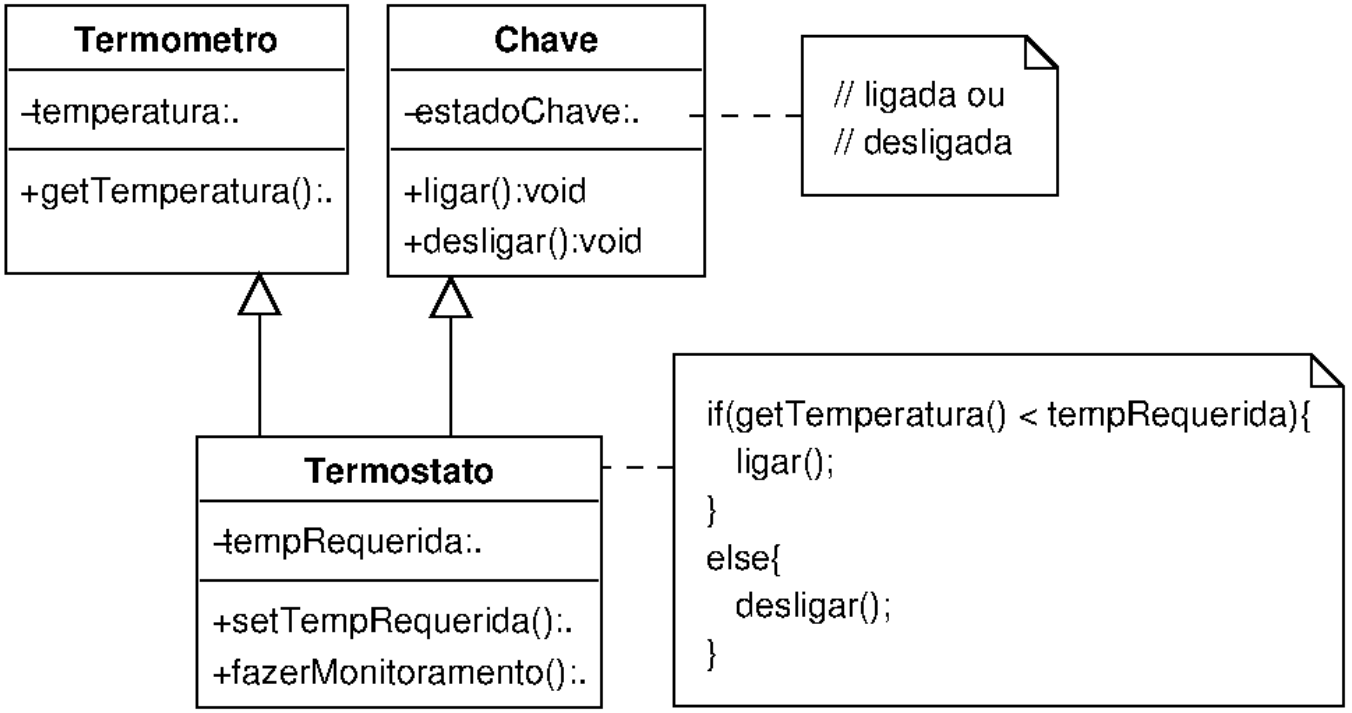
Redefinição de Operações

- permite que uma subclasse implemente uma implementação específica de um método já definido por alguma de suas superclasses.
- a implementação na subclasse redefine/substitui (*overrides*) a implementação da superclasse quando a subclasse proporciona um método com a mesma assinatura da superclasse.
- O método redefinido na subclasse tem o mesmo nome da superclasse, os tipos dos parâmetros são os mesmos apresentados na mesma ordem e no mesmo número, e o tipo de retorno é idêntico.

Exemplo de Herança Múltipla (I)

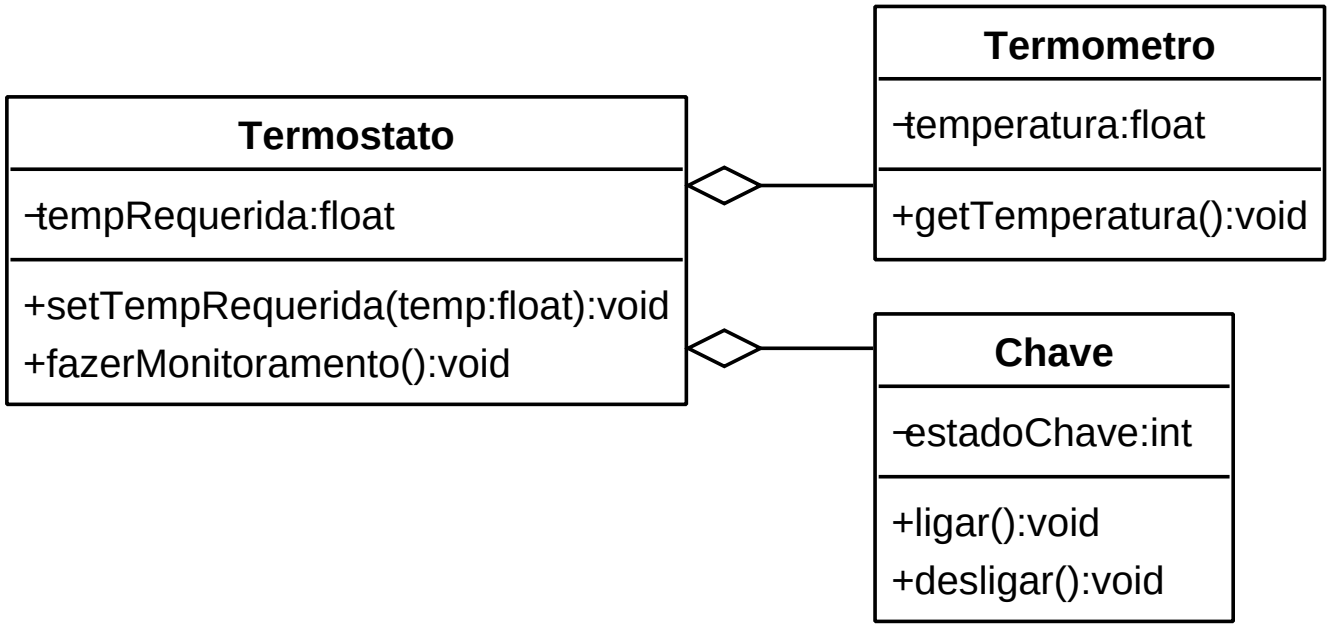
- Termostato: dispositivo que mantém um sistema numa temperatura constante
- Um termostato envolve 2 elementos: um termômetro e uma chave
- Um Termostato pode ser visto como um tipo de termômetro e como um tipo de chave
- Um termostato mantém um sistema numa temperatura constante. A chave pode estar conectada a um sistema de aquecimento central e irá desligar/ligar o sistema conforme a temperatura desejada

Exemplo de Herança Múltipla (II)




```
Termometro t = new Termostato();
Chave c = new Termostato();

Termostato ter = new Termostato();
```

Solução Usando Agregação




```
Termometro t = new Termostato(); // ERRO
Chave c = new Termostato(); // ERRO
```

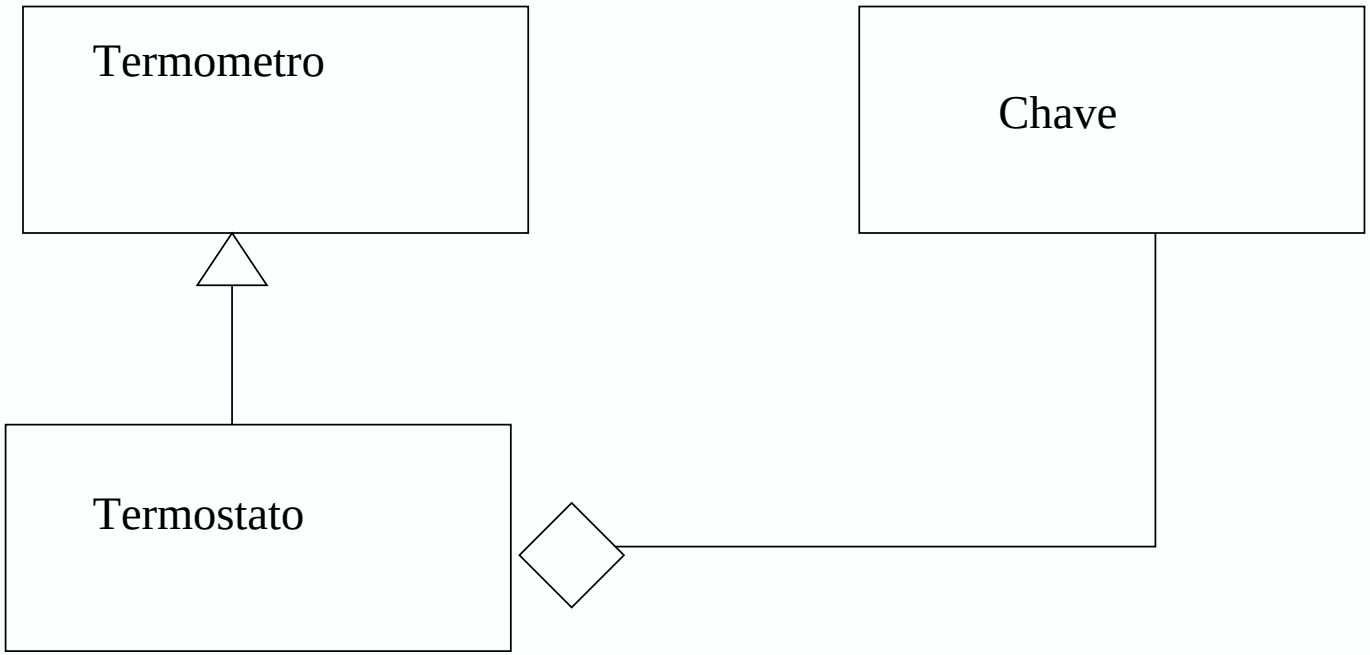
Solução Usando Agregação (I)

```
class Termometro{
   private float temperatura;

   public float getTemperatura(){
      return temperatura;
   } }

class Chave{
   private int estadoChave = 0; // 0-desligado; 1-ligado.

   public void ligar(){
      estadoChave = 1; }

   public void desligar(){
      estadoChave = 0; } }
```

Solução Usando Agregação (II)

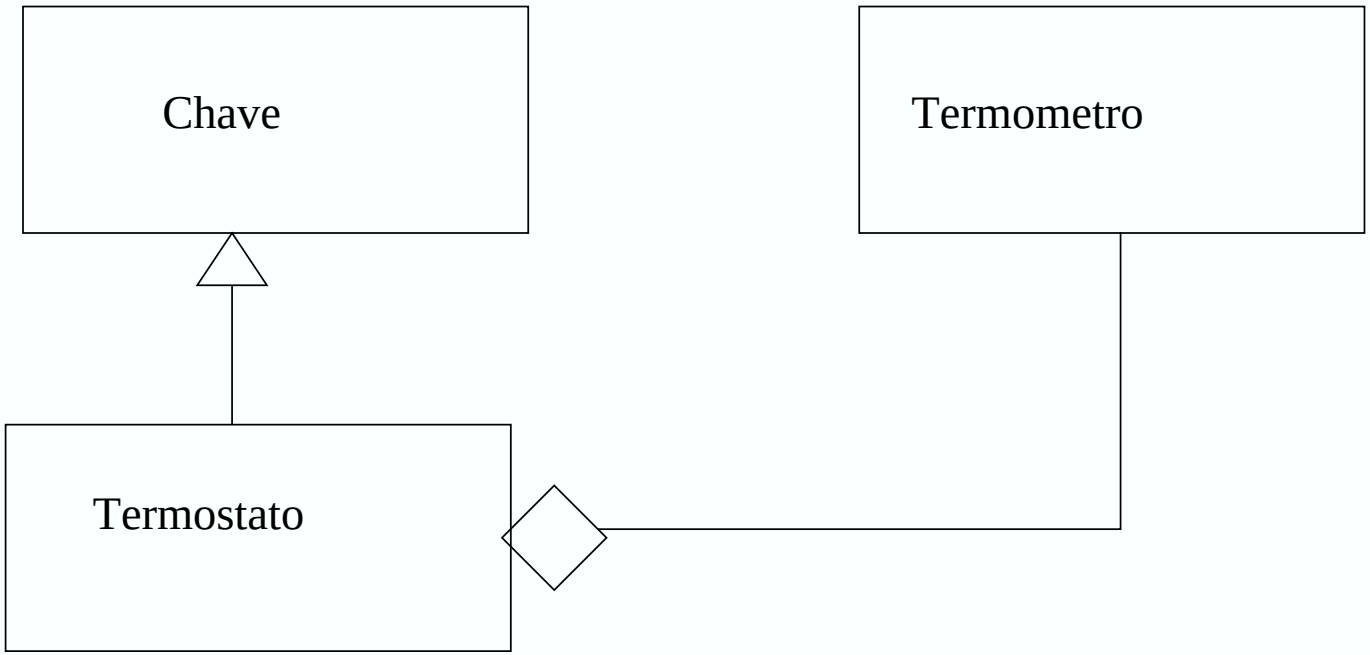
```
class Termostato{
  private Termometro termometro;
  private Chave chave;
  private float tempRequerida;

  public Termostato(){ // CONSTRUTOR
    termometro = new Termometro();
    chave = new Chave(); } //FIM-CONSTRUTOR
  public void setTempRequerida(float temp){
    tempRequerida = temp; } //FIM-SETTEMPREQUERIDA
  public void fazerMonitoramento(){
    if (termometro.getTemperatura()>temperaturaRequerida)
       chave.desligar();
    else
       chave.ligar(); }
```

Solução Usando Herança Simples e Agregação (I)




```
Termometro t = new Termostato(); // OK
Chave c = new Termostato(); // ERRO
```

Solução Usando Herança Simples e Agregação (II)



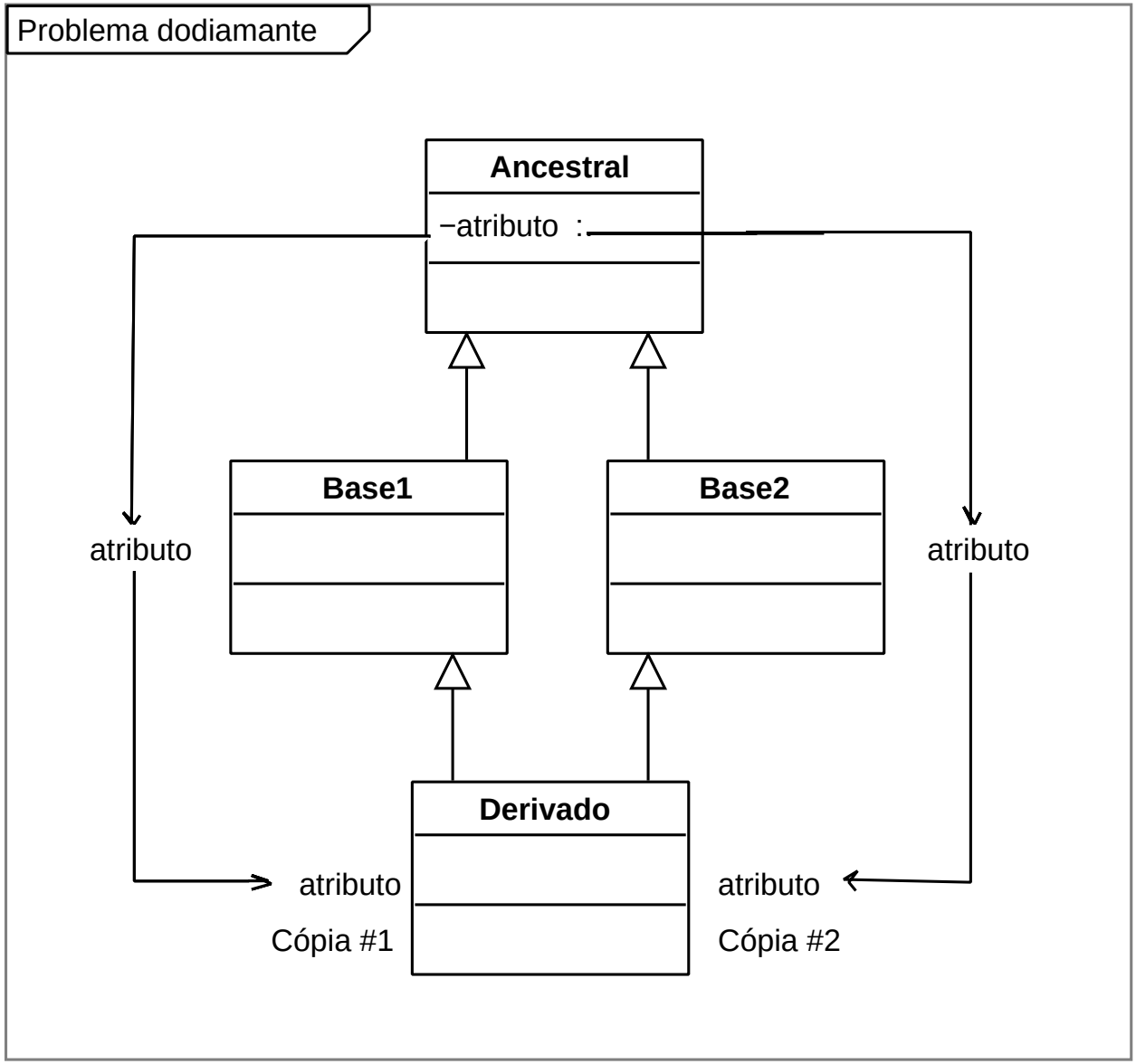

```
Termometro t = new Termostato(); // ERRO
Chave c = new Termostato(); // OK
```

Quando usar Herança Múltipla

- Se a nova abstração é um tipo de alguma outra abstração, ou se o comportamento da nova abstração é exatamente igual à soma das suas componentes, então herança múltipla é mais apropriada.
- Se a nova abstração é maior, em termos de comportamento, do que a soma do comportamento das abstrações bases, então agregação é mais apropriada.

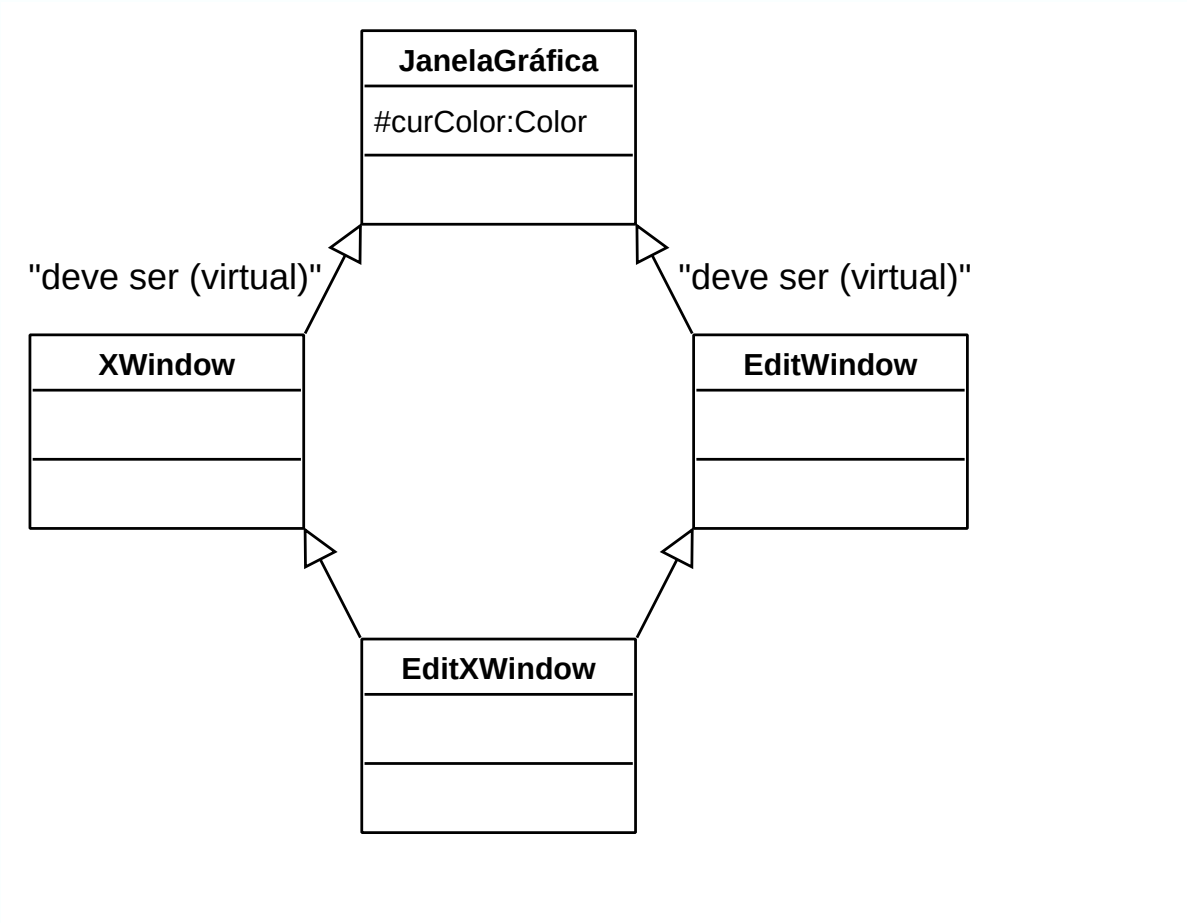
Problema do Diamante (I)

- O **problema do diamante** é caracterizado pela ambigüidade causada pela herança repetida de atributos oriunda de uma classe ancestral comum.
- Os membros da classe ancestral são herdados por cada uma das classes intermediárias.
- A classe derivada que herda de múltiplas classes intermediárias possue atributos replicados em endereços de memória distintos.


Problema dodiamante
Ancestral
−atributo :
Base1
Base2
atributo
atributo
Derivado
atributo
Cópia #1
atributo
Cópia #2

Solução: Problema do Diamante (II)

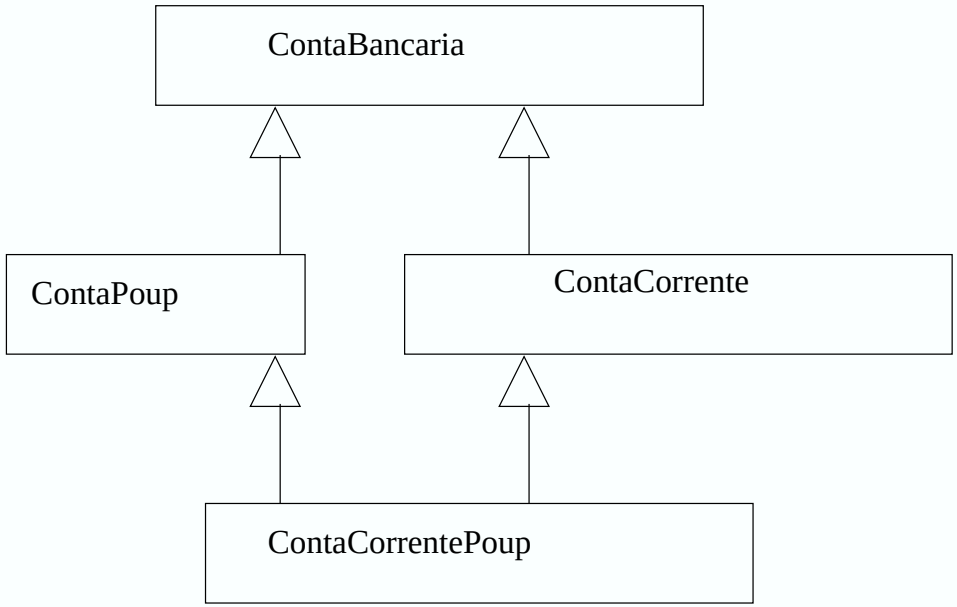
As classes que herdam do ancestral comum devem usar o modificador **virtual** no relacionamento de herança.



Problema do Diamante (III)

```
class JanelaGrafica{
  public:
    Window();
    ...
  protected:
    Color curColor;
}

class XWindow : public virtual JanelaGrafica{...}

class EditWindow : public virtual JanelaGrafica{...}

class EditXWindow : public XWindow, public EditWindow{...}
```

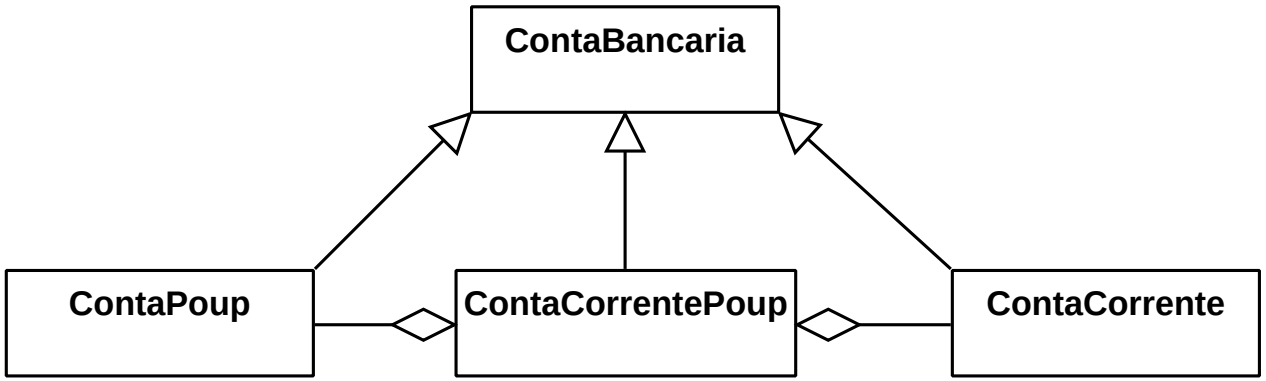
Refactoring de Herança Múltipla (I)



Princípio da Substituição: **ContaCorrentePoup** é um tipo de **ContaPoup**, de **ContaCorrente** e de **ContaBancaria**.

```
ContaBancaria cb = new ContaCorrentePoup(); // OK
ContaPoup cp = new ContaCorrentePoup();   // OK
ContaCorrente cc = new ContaCorrentePoup(); // OK
```

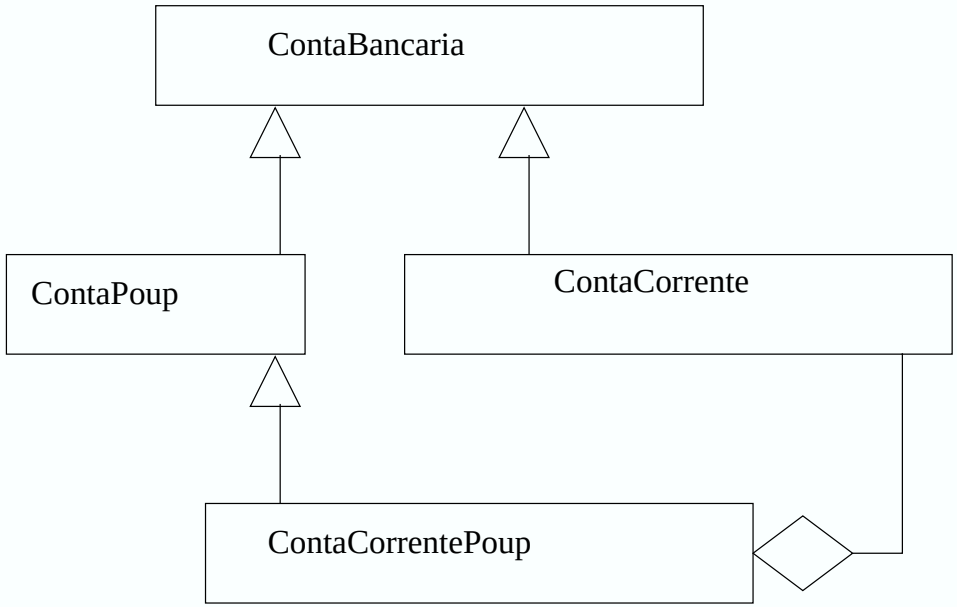
Refactoring de Herança Múltipla (II)



Perda Parcial do Princípio da Substituição: **ContaCorrentePoup** é um tipo de **ContaBancaria**, mas ela não é um tipo de **ContaPoup** e nem **ContaCorrente**

```
ContaBancaria cb = new ContaCorrentePoup(); // OK
ContaPoup cp = new ContaCorrentePoup(); // ERRO
ContaCorrente cc = new ContaCorrentePoup(); // ERRO
```

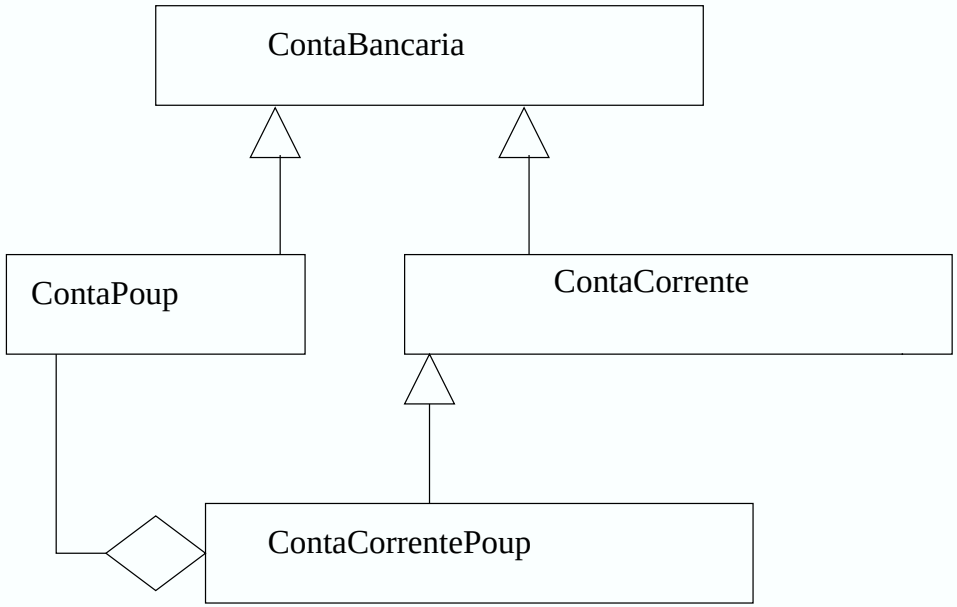
Refactoring de Herança Múltipla (III)



Perda: **ContaCorrentePoup** é um tipo de **ContaBancaria** e tb é um tipo de **ContaPoup**, mas não é um tipo de **ContaCorrente**.

```
ContaBancaria cb = new ContaCorrentePoup(); // OK
ContaPoup cp = new ContaCorrentePoup(); // OK
ContaCorrente cc = new ContaCorrentePoup(); // ERRO
```

Refactoring de Herança Múltipla (IV)

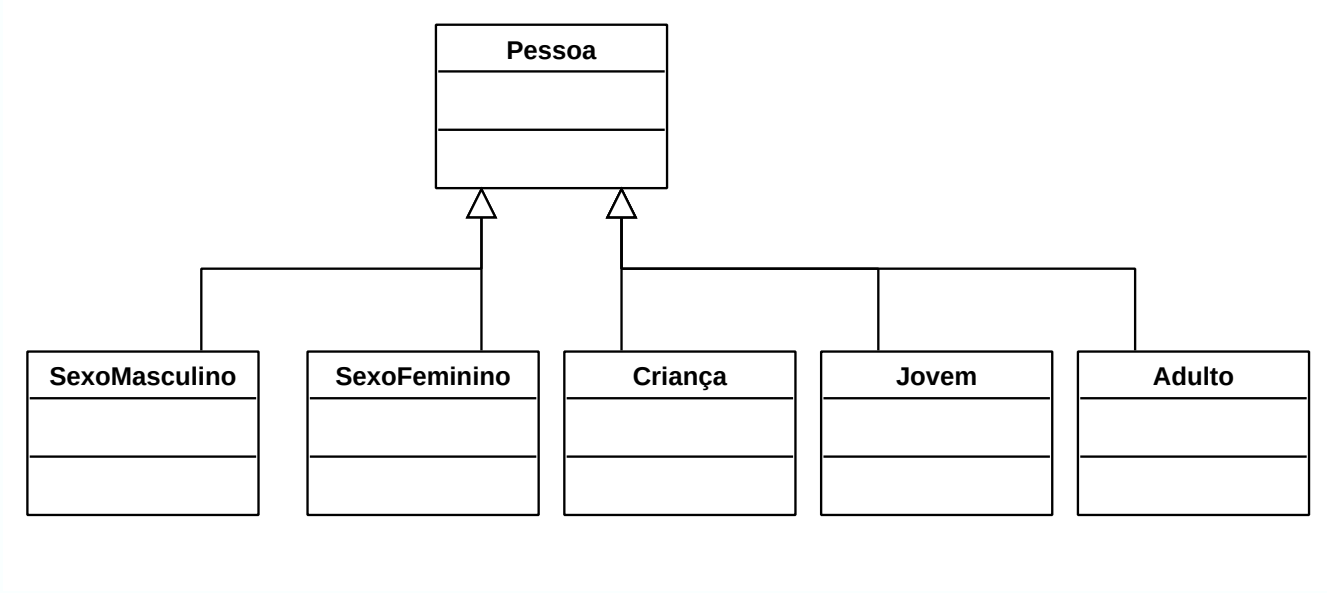


Perda: **ContaCorrentePoup** é um tipo de **ContaBancaria** e um tipo de **ContaCorrente**, mas não é um tipo de **ContaPoup**.

```
ContaBancaria cb = new ContaCorrentePoup(); // OK
ContaPoup cp = new ContaCorrentePoup(); // ERRO
ContaCorrente cc = new ContaCorrentePoup(); // OK
```

Exercício 1

- Uma abstração pode ser vista através de múltiplas perspectivas. Por exemplo, uma pessoa pode ser classificada de acordo com o seu sexo como masculino ou feminino e pode também ser classificada de acordo com a sua idade como criança, jovem e adulto.

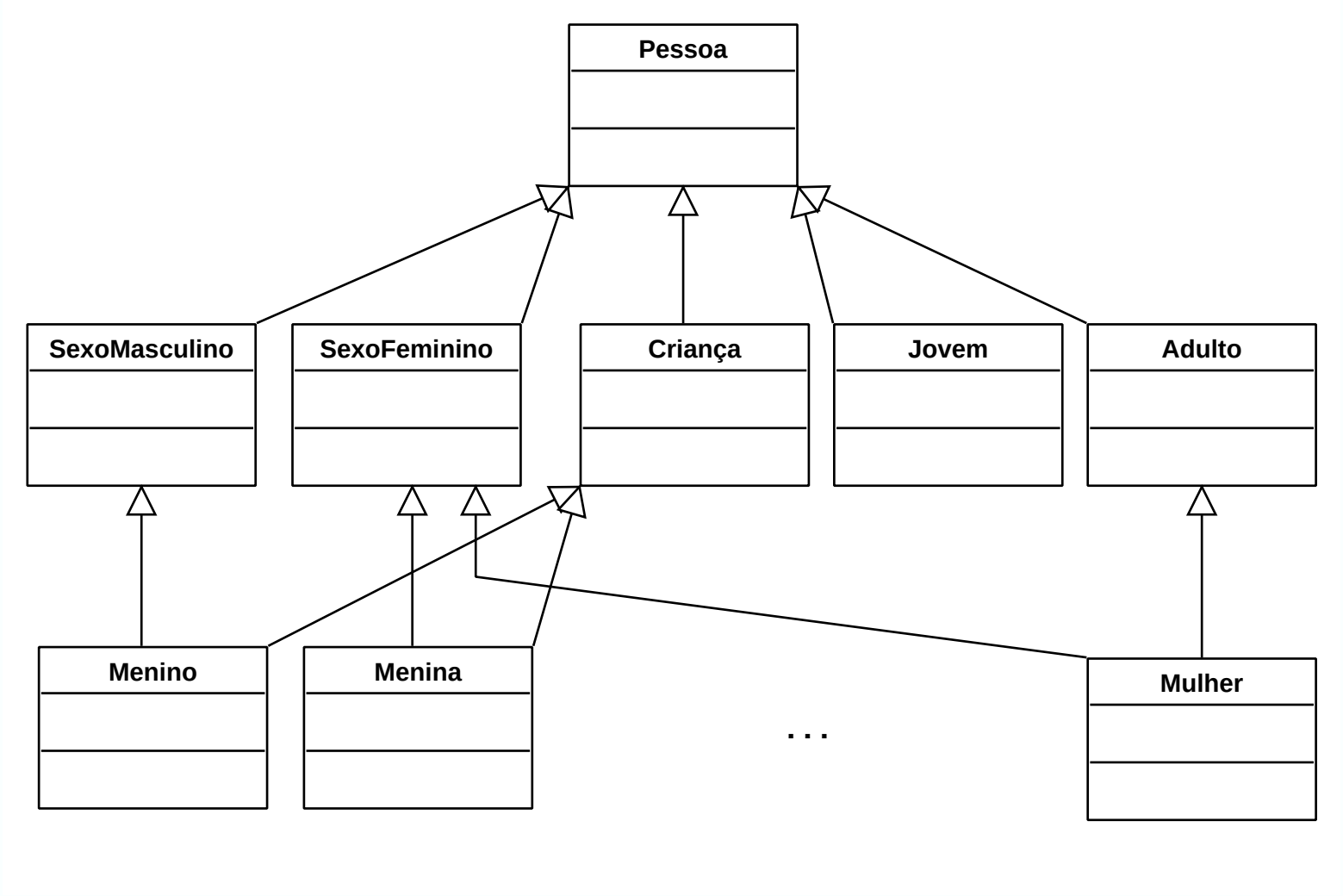


Exercício 1 (Cont.)

1. Apresente uma solução de modelagem que combine essas duas perspectivas usando herança múltipla. Note que um objeto pode ser instanciado a partir de apenas uma única classe (i.e., um objeto não pode ser instância de 2 classes). Por exemplo, o objeto José é do sexo masculino e adulto, mas ele não pode ser instanciado a partir de ambas as classes SexoMasculino e Adulto ao mesmo tempo.
2. Proponha uma solução alternativa mais flexível que supere as desvantagens identificadas no item anterior. (Dica: pense na hierarquia de agregação/decomposição.)

Respostas

Resposta – Exercício 1 (a)



Resposta – Exercício 1 (b)