Delegação

Compartilhamento de comportamento

- Compartilhamento de comportamento pode ser obtido pelo menos de duas formas diferentes:
  - Compartilhamento baseado em classes (mecanismo de herança)
  - Compartilhamento baseado em protótipos/objetos (mecanismo de delegação)

Sistemas Baseados em Delegação (I)

- Nessas linguagens, sistemas baseados em delegação, em geral, não têm como foco principal a estruturação baseada em classes
- Os objetos são vistos como **protótipos** ou **exemplares**
- Esses protótipos podem assumir dois papéis, que são definidos dinamicamente:
  - **delegadores** – delegam seu comportamento para outros protótipos
  - **delegados** – executam o comportamento dos seus delegadores utilizando o estado do delegador inicial

Sistemas Baseados em Delegação (II)

- Com a utilização de delegação, alterações efetuadas nos métodos e estrutura de uma classe, são automaticamente passadas para todos os seus delegados
- Nesses sistemas, as distinções entre classes e objetos são eliminadas, ficando apenas a noção de objetos (protótipos)
- O desenvolvedor inicialmente identifica um conjunto de protótipos, e depois descobre similaridades e/ou diferenças com outros objetos
- Dentro dessa nova concepção, um objeto antigo pode tornar-se um protótipo para um objeto recém-descoberto. A idéia é começar com casos particulares, e depois generalizá-los ou especializá-los

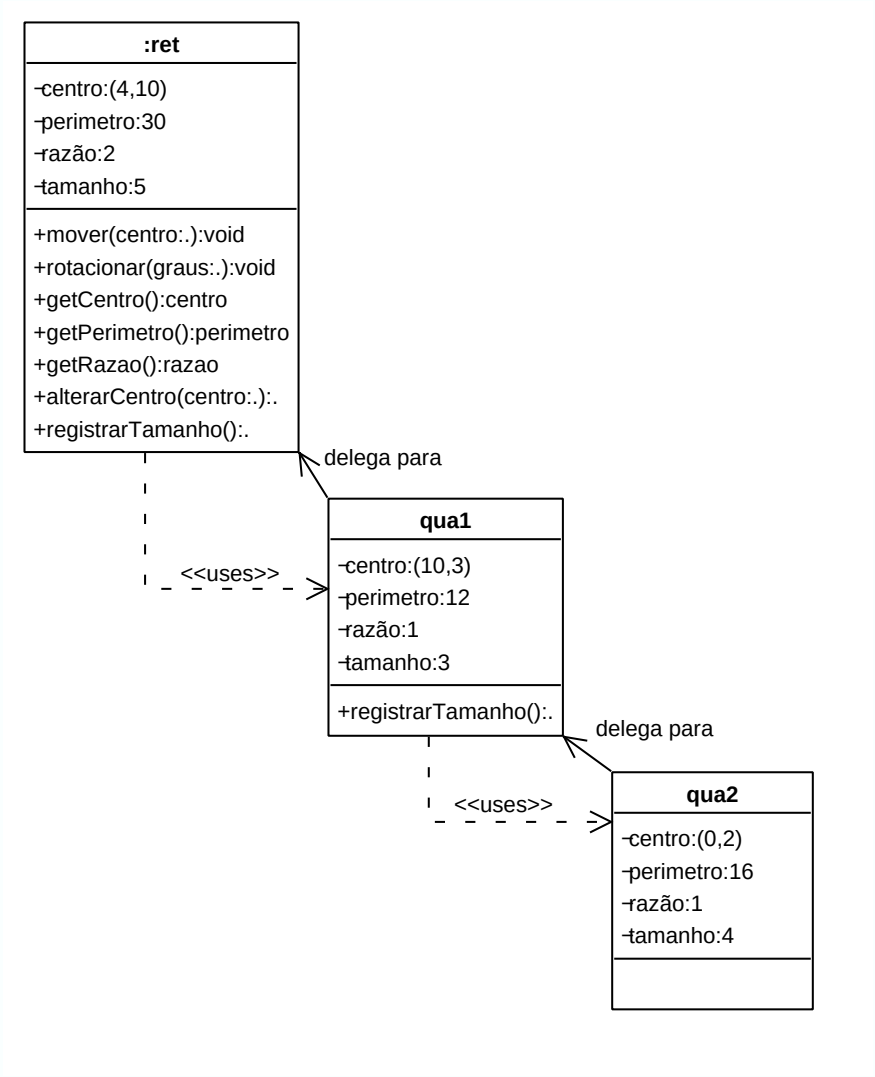
Exemplo 1 (I)

- Suponha que um desenvolvedor inicialmente identifique um objeto retangular, e, em seguida, ele identifique um objeto quadrado
- Nesse caso, podemos dizer que o quadrado "se parece"com um retângulo
- Suponha ainda que posteriormente o desenvolvedor identifique um outro quadrado (maior do que o primeiro)

Exemplo 1 (II)

- Num sistema baseado em protótipos, podemos definir o retângulo como sendo o protótipo para a criação do primeiro quadrado
- Similarmente, o primeiro quadrado pode ser definido como o protótipo de criação para o segundo quadrado

Exemplo 1 (III)



Exemplo 1 (IV)

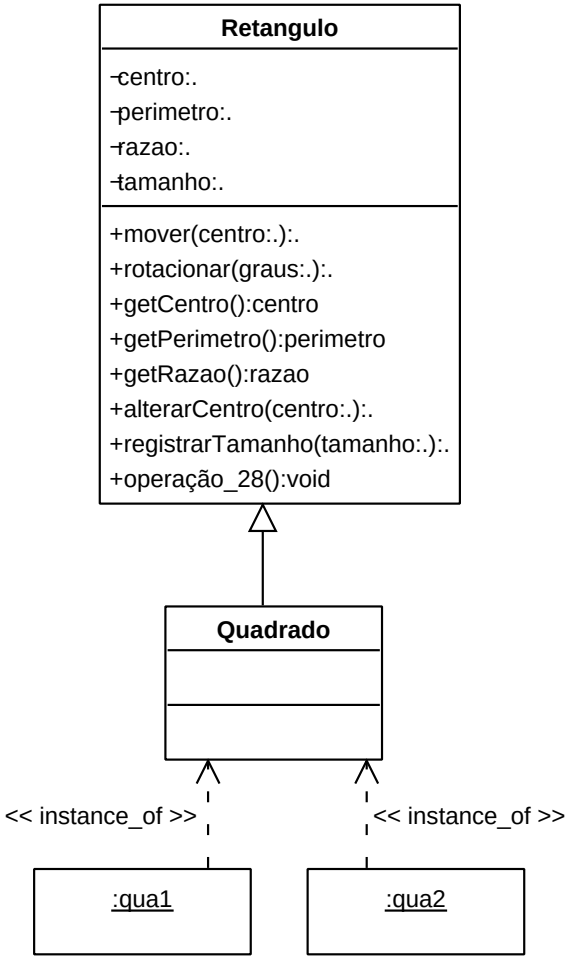
- O objeto qua1 tem uma associação de **delega-para** com o objeto ret. Isto significa que os atributos e operações que não são redefinidos em qua1 serão "herdados" de ret.
- O fato da operação reajustarTamanho() ter sido redefinida no protótipo qua1 significa que sua execução não será delegada para ret

Exemplo 1 (V)

- Portanto, qua1 delega a execução de mensagens, como por exemplo rotacionar(), para o seu objeto delegado ret. No entanto, vale ressaltar que quando a operação rotacionar() é encontrada no protótipo ret, essa operação é executada no estado do objeto qua1, isto é, o receptor inicial da mensagem

Exemplo 1 (VI) - Modelo de Classes

- É possível remodelar a solução utilizando o modelo de classes:



Delegação X Herança(I)

- O mecanismo de **herança** permite compartilhamento de comportamento, isto é, reutilização de operações, entre classes derivadas e classes bases (estabelecido estaticamente)
- Esse mecanismo não é aplicado a objetos individualmente
- O mecanismo de **delegação** também permite compartilhamento de comportamento, mas opera diretamente entre objetos, não entre classes (estabelecido dinamicamente)

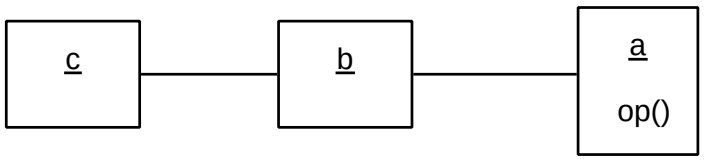
Delegação X Herança(II)

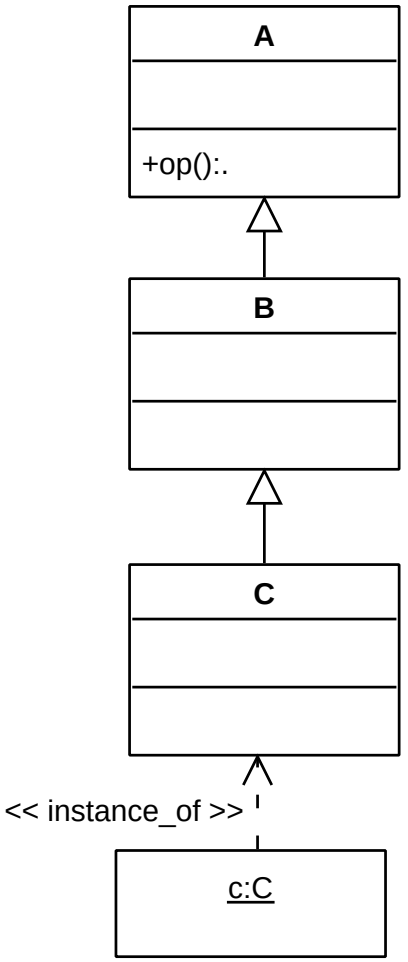
- Portanto, linguagens baseadas em classes usam um compartilhamento de comportamento estático e baseado em grupos, contrastando com as linguagens baseadas em delegação que proporcionam um compartilhamento de comportamento dinâmico e baseado em protótipos (objetos).
- O mecanismo de delegação é mais genérico do que o mecanismo de herança, no sentido que o mecanismo de delegação pode simular o mecanismo de herança, mas não vice-versa

Delegação X Herança(III)

- Em sistemas baseados em delegação, a lista de delegados de um protótipo pode conter zero ou muitos delegados e ser dinamicamente mudada
- A principal vantagem de delegação sobre herança é que delegação facilita a mudança de implementação de operações em tempo de execução

Delegação X Herança(IV)





Delegação em Sistemas Baseados em Classes

Delegação em Sistemas Baseados em Classes (I)

- Embora herança e delegação sejam geralmente definidas como soluções alternativas no projeto de sistemas orientados a objetos, delegação pode ser "simulada" em linguagens baseadas em classes de forma ad hoc
- Essa implementação de delegação em sistemas baseados em herança é uma forma de implementar compartilhamento de comportamento quando um objeto precisa, por exemplo, ser capaz de alterar suas respostas para pedidos de serviços em tempo de execução

Delegação em Sistemas Baseados em Classes (II)

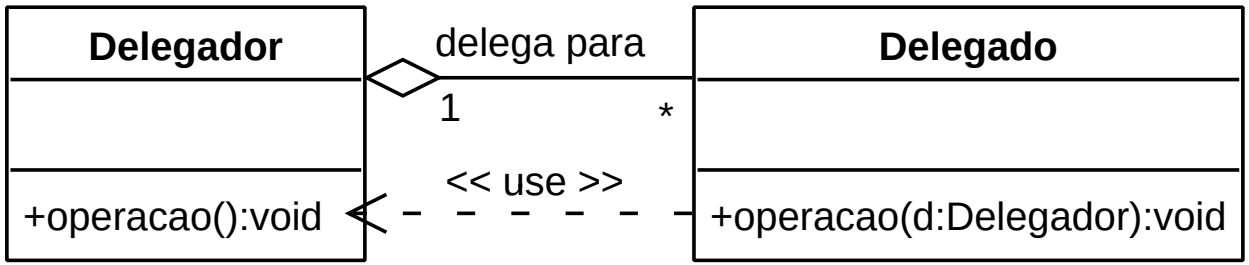
- Delegação pode ser vista como uma técnica de projeto alternativa que pode ser usada em qualquer linguagem orientada a objetos baseada em classes, incluindo aquelas que são estaticamente tipadas

O Padrão Delegação (I) - Motivação

- Em sistemas baseados em classes, uma vez que um objeto é criado, ele está permanentemente ligado ao comportamento estabelecido na definição do seu tipo
- Mas muitas entidades do mundo real exibem diferentes tipos de comportamentos em diferentes estágios de vida
- Além disso, muitas vezes o desenvolvedor gostaria de reutilizar comportamentos já implementados por outros protótipos sem ter a obrigação de garantir que o relacionamento de generalização/especialização seja verdadeiro

O Padrão Delegação (II) - Solução

- Delegação pode ser implementada numa linguagem baseada em classes através da inclusão do receptor original (isto é, o objeto delegador) como um parâmetro extra para cada mensagem delegada para o objeto delegado



O Padrão Delegação (III) - Conseqüências

- A implementação do padrão de delegação numa linguagem baseada em classe requer, portanto, um esforço extra por parte do desenvolvedor pois:
  - O objeto delegador e os seus objetos delegados devem ser estaticamente definidos (isto é, não existiria criação dinâmica de delegados a princípio)
  - Um objeto delegado ao receber uma mensagem delegada, utiliza o parâmetro extra com a referência do receptor inicial da mensagem para realizar operações no estado do delegador

O Padrão Delegação (IV) - Conseqüências

- Essa é uma estrutura recursiva, uma vez que o objeto delegado pode assumir o papel de delegador e ter sua própria lista de delegados, e assim por diante
- Linguagens baseadas em delegação implementam este nível extra de indireção de forma transparente
- Por enfatizarem principalmente a flexibilidade do sistema, linguagens baseadas em delegação dão suporte para mudanças em tempo de execução baseando-se em verificação dinâmica de tipos

O Padrão de Projeto State

O Padrão de Projeto State

- O padrão de projeto State permite que um objeto altere seu comportamento quando seu estado interno é alterado
- Ele usa o mecanismo de delegação e o relacionamento de agregação para lidar com situações em que um objeto precisa se comportar de maneiras diferentes em diferentes circunstâncias

O Problema (I)

- Quando nós consideramos o comportamento de entidades no mundo real, nós observamos que elas geralmente têm um **ciclo de vida**
- Muitas vezes entidades no mundo real exibem diferentes fases de comportamento durante o seu ciclo de vida.
  - Por exemplo, uma borboleta começa a vida como uma caterpilar, depois muda para uma crisálida e finalmente se transformam numa borboleta adulta

O Problema (II)

- O termo **estado lógico** é aplicado para cada fase de comportamento observável da borboleta, isto é, caterpilar, crisálida e borboleta adulta
- Num dado momento, diferentes instâncias de uma classe podem estar em estados lógicos diferentes. O estado lógico de uma instância em particular é chamado de **estado lógico corrente**
- Em algumas situações, o comportamento de um objeto depende do seu estado e esse comportamento precisa ser modificado em tempo de execução, quando esse estado interno muda

O Problema (III)

- Normalmente, esse problema é resolvido através de métodos que usam comandos condicionais para decidir qual comportamento deve ser adotado, dependendo do estado do objeto
- Essa abordagem não é recomendada por dois motivos principais:
  - O código pode ser repetido em vários métodos (tanto código relativo às condições quanto ao comportamento)
  - Além disso, um número grande de condições complexas torna o código dos métodos muito difícil de entender e manter.

Exemplo 2 - Sistema Bancário

Suponha que um banco tem apenas um tipo de conta e que existe uma política restrita sobre essas contas de tal forma que o banco não permite que os seus saldos fiquem negativo. Além disso, uma conta é considerada normal se ela mantém um saldo positivo maior que zero. Se o saldo da conta é zerado, uma conta normal é transferida para uma conta anormal, onde a retirada de dinheiro fica suspensa, ficando somente permitido transações de depósito em conta. Se uma conta anormal tem um saldo positivo então ela é automaticamente revertida para uma conta normal.

Exemplo 2 - Primeira Solução (I)

- Uma solução possível é representar os estados lógicos de um objeto Conta dentro dos atributos da classe Conta
- Mudanças nos valor desses atributos são detectados por comandos condicionais

```
class Conta {
   double saldo;
   int flagNormal; // 0=abnormal 1=normal
   ...
```

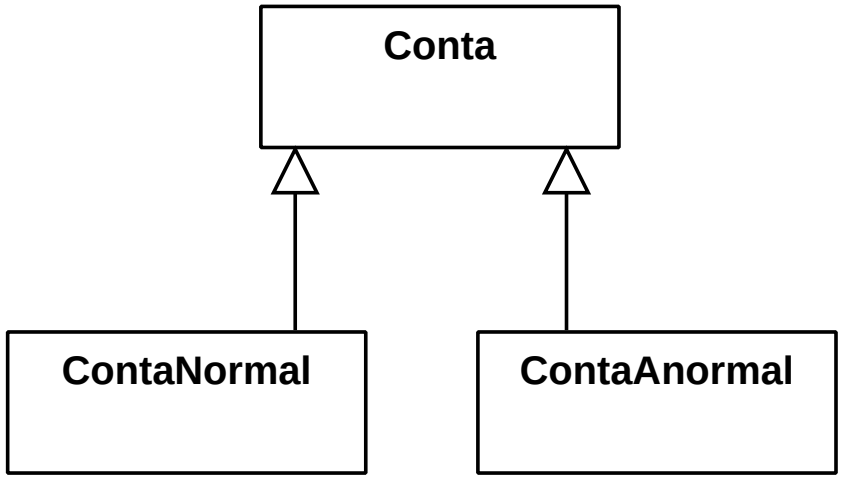
Exemplo 2 - Primeira Solução (II)

```
   ...
   public debitar(float qtde){
      (...)
      if (flagNormal = 1) { //realizar o saque
      }
      else { // imprimir uma mensagem de aviso:
             // transação de débito suspensa
      }
   } // fim do método debitar()
   public creditar(float qtde){
      // realizar o crédito
      if ((flagNormal = 0) and (saldo > 0)){
         flagNormal = 1;}
   } // fim do método creditar()
} // fim da classe Conta
```

Exemplo 2 - Segunda Solução (I)

- Uma solução alternativa é eliminar os comandos condicionais inseridos na implementação das operações da classe Conta
- A criação de tipos ContaNormal e ContaAnormal substitui o atributo flagNormal que guarda uma informação de tipo na proposta da primeira implementação

Exemplo 2 - Segunda Solução (II)



Exemplo 2 - Segunda Solução (III)

```
class Conta {
  double saldo;
  public abstract debitar(float qtde){}
  public creditar(float qtde){// realizar o crédito}
}

class ContaNormal extends Conta {
  public debitar(float qtde){
    //  realizar o saque
    if (saldo = 0) {
          // transição de ContaNormal para ContaAnormal,
          // eliminando o objeto do tipo
          // ContaNormal e criando um objeto ContaAnormal
    }
  }}
```

Exemplo 2 - Segunda Solução (IV)

```
class ContaAnormal extends Conta {
  public debitar(float qtde){
    // imprimir uma mensagem de aviso:
    // transação de débito suspensa
  }
  public creditar(float qtde){
    creditar();
    if (saldo > 0){
          // transição de ContaAnormal para ContaNormal,
          // eliminando o objeto do tipo
          // ContaAnormal e criando um objeto ContaNormal
    }
  }
}
```

Exemplo 2 - Segunda Solução (V)

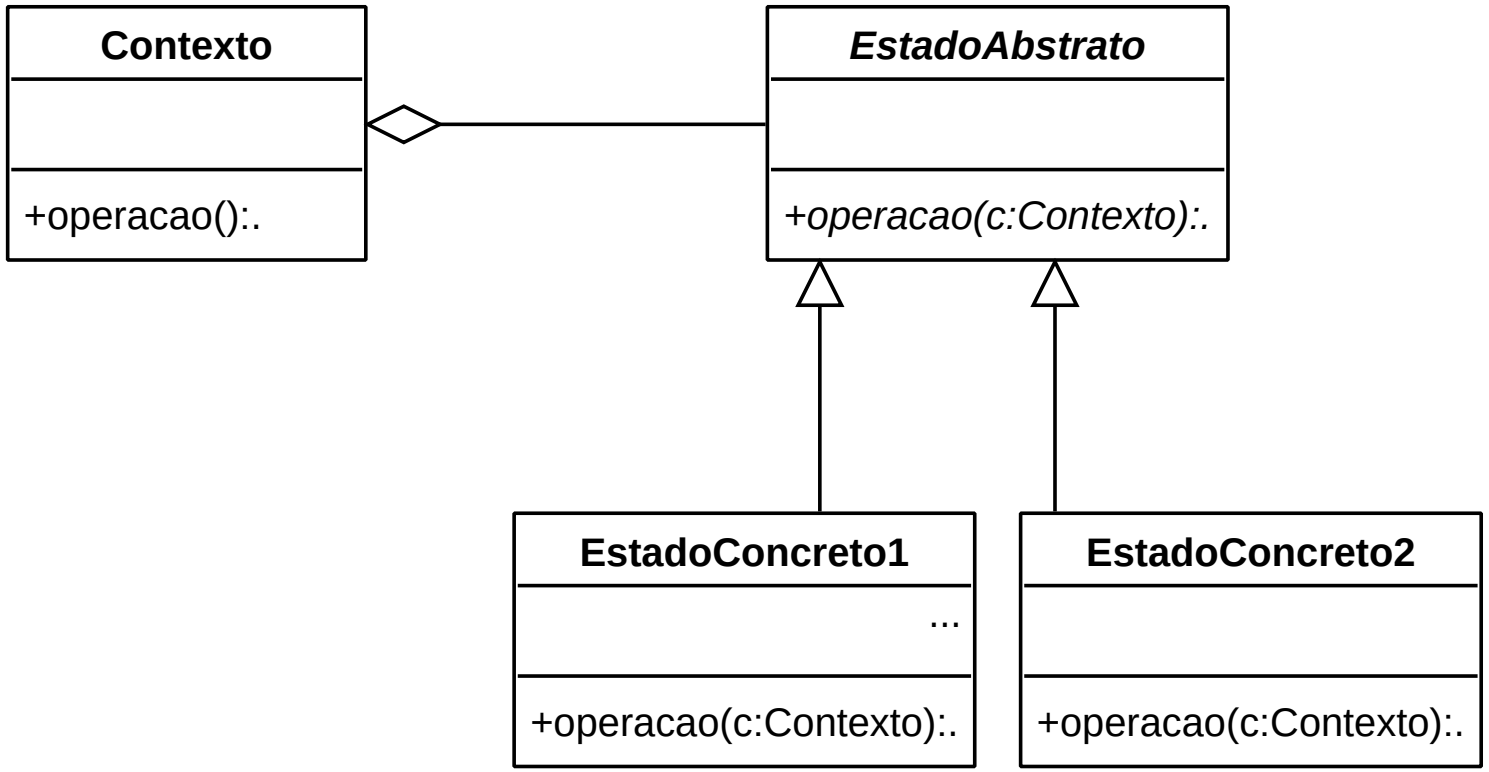
- Nessa solução, a operação DEBITAR() está redefinida nas classes CONTANORMAL e CONTAANORMAL
- Ela captura o relacionamento entre as diferentes informações sobre uma conta, mas ela tem pelo menos duas limitações:
  1. Linguagens orientadas a objetos da escola escandinava, como Java e C++, não permitem que objetos mudem de classes
  2. Isto implica quando um objeto conta muda de estado, um novo objeto deve ser criado e o antigo objeto deve ser destruído (difícil gerenciamento)
  3. As classes CONTANORMAL e CONTAANORMAL não são realmente subtipos de conta, mas sim estados conceituais separados pertencentes a uma mesma abstração

Exemplo 2 - Terceira Solução (I)

O Padrão State:

- Uma terceira solução é criar classes que encapsulam os diferentes estados lógicos do objeto conta e os comportamentos associados a esses estados
- Dessa forma, a complexidade do objeto é quebrada em um conjunto de objetos menores e mais coesos

Exemplo 2 - Terceira Solução (II)



Exemplo 2 - Terceira Solução (III)

- A classe Contexto define objetos cujos estados são decompostos em diversos objetos diferentes definidos por subtipos da classe abstrata Estado
- A classe Estado define uma interface unificada para encapsular o comportamento associado a um estado particular de de um objeto do tipo Contexto
- Um objeto do tipo Contexto mantém uma referência para uma instância de uma subclasse concreta de Estado
- Para cada estado de Contexto, é definida uma subclasse EstadoConcreto de Estado que implementa o comportamento associado a esse estado

Exemplo 2 - Terceira Solução (IV)

- Um objeto CONTEXTO delega requisições dependentes de estado para o objeto ESTADOCONCRETO atual, de acordo com o Padrão Delegação
- A classe CONTEXTO passa uma referência de si próprio como argumento para o objeto do tipo ESTADO responsável por tratar a requisição
- Clientes podem configurar um contexto com objetos do tipo ESTADO
- Uma vez que isso seja feito, porém, esses clientes passam a se comunicar apenas com o objeto do tipo CONTEXTO e não interferem mais com seu estado

Conseqüências

- Localiza o comportamento dependente de estado e particiona o comportamento relativo a diferentes estados. O padrão substitui uma classe Contexto complexa por uma classe Contexto muito mais simples e um conjunto de subclasses de Estado altamente coesas
- Torna transições de estado explícitas, ao invés de espalhá-las em comandos condicionais
- Torna o estado de um objeto compartilhável, caracterizando uma quebra de encapsulamento entre a classe Contexto e a classe Estado.

Exemplo 2 - Terceira Solução (V)

```
//arquivo Conta.java
package conta; //classe pertencente ao pacote ‘‘conta’’
class Conta {
 double saldo; // visibilidade de pacote
 private EstConta estado;
 private EstContaNormal estContaNormal;
 private EstContaAnormal estContaAnormal;
 ...
```

Exemplo 2 - Terceira Solução (VI)

```
...
 public Conta(){
  estNormal = new EstContaNormal();
   estAnormal = new EstContaAnormal();
   estado = estNormal;  }   //fim do construtor
 void putEstNormal(){estado = estNormal;}
 void putEstAnormal(){estado = estAnormal;}
 public void debitar(float qtde){
    estado.debitar(qtde,this);}
 public void creditar(float qtde){
    estado.creditar(qtde,this);}}
```

Exemplo 2 - Terceira Solução (VII)

```
//arquivo EstConta.java
package conta;
abstract class EstConta { // visibilidade de pacote
 abstract void debitar(float qtde,Conta c); // pacote
 abstract void creditar(float qtde,Conta c); // pacote}
//arquivo EstContaNormal.java
package conta;
class EstContaNormal extends EstConta { // vis. de pacote
 void debitar(float qtde,Conta c){ // vis. de pacote
    c.saldo = c.saldo - qtde;
    if (c.saldo <= 0) {
      //transição de EstContaNormal para EstContaAnormal,
      c.putEstAnormal();}
 void creditar(float qtde,Conta c){ // vis. de pacote
    c.saldo = c.saldo + qtde;} }
```

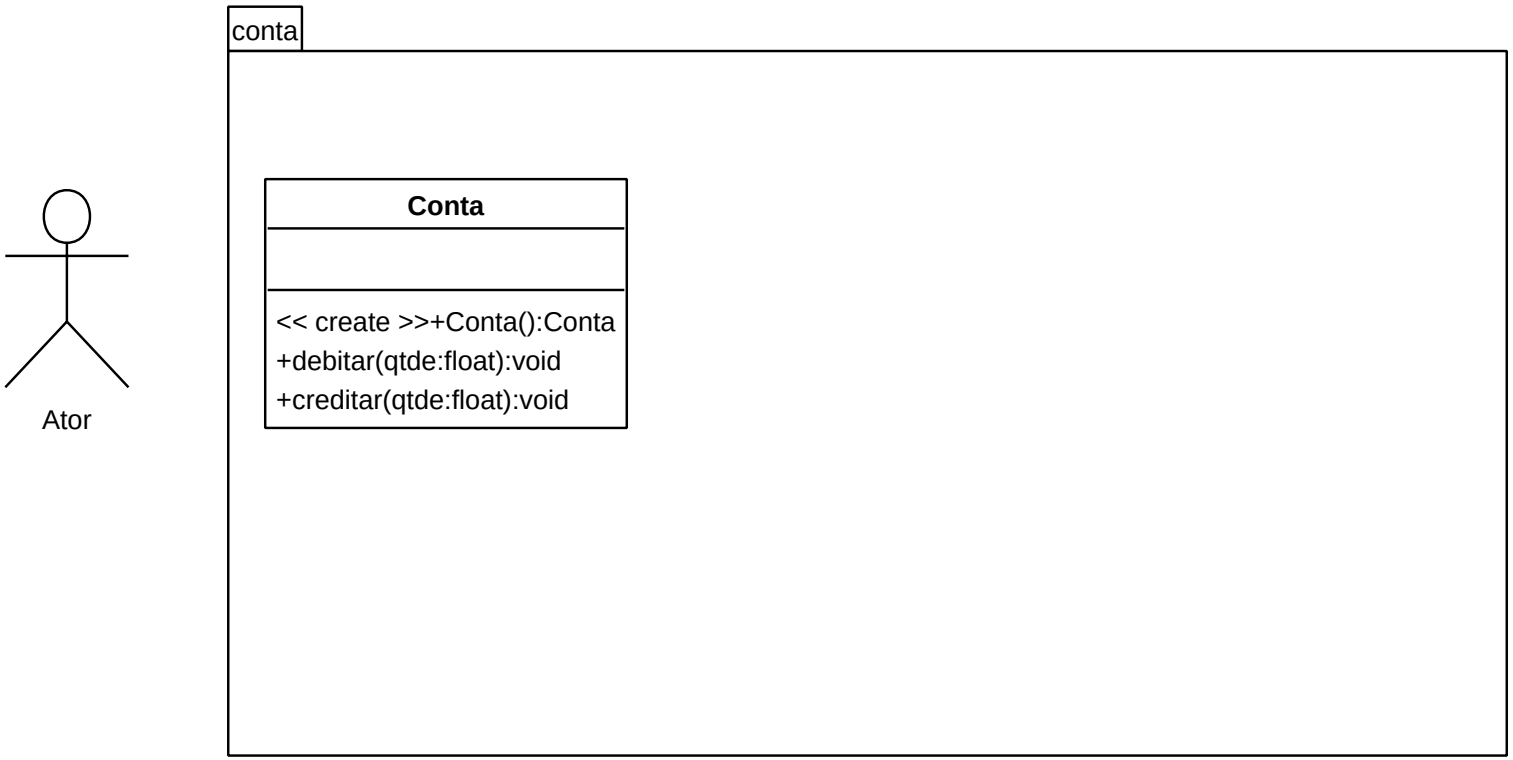
Exemplo 2 - Terceira Solução (VIII)

```
//arquivo EstContaAnormal
package conta;
class EstContaAnormal extends EstConta { // vis. de pacote

 void debitar(float qtde,Conta c){ // vis. de pacote
   // imprimir uma mensagem de aviso: transação de
   // débito suspensa
   System.out.println(‘‘transação de débito suspensa’’); }
 void creditar(float qtde,Conta c){ // vis. de pacote
   c.saldo = c.saldo + qtde;
   if (c.saldo > 0){
     //transição de EstContaAnormal para EstContaNormal,
     c.putEstNormal(); }
 } }
```

Modelagem do Sistema Bancário

Visão externa ao módulo:



Modelagem do Sistema Bancário

Visão interna ao módulo: