Solução ao Pedido 5:

```
1  // Withdrawal.cs
2  // Class Withdrawal represents an ATM withdrawal transaction
3  public class Withdrawal
4  {
5     // attributes
6     private int accountNumber; // account to withdraw funds from
7     private decimal amount; // amount to withdraw
8
9     // references to associated objects
10    private Screen screen; // ATM's screen
11    private Keypad keypad; // ATM's keypad
12    private CashDispenser cashDispenser; // ATM's cash dispenser
13    private BankDatabase bankDatabase; // account-information database
14
15    // parameterless constructor
16    public Withdrawal()
17    {
18       // constructor body code
19    } // end constructor
20

21    // operations
22    // perform transaction
23    public void Execute()
24    {
25       // Execute method body code
26    } // end method Execute
27 } // end class Withdrawal
```

Incorporando herança e polimorfismo ao sistema ATM

Para aplicar a herança, procure aspectos comuns entre as classes no sistema.
Crie uma hierarquia de herança para modelar classes semelhantes (mas não idênticas) de uma maneira mais elegante e eficiente.
Modifique o diagrama de classes para incorporar as novas relações de herança e traduza o design atualizado para código Java.
Problema de representar uma transação financeira no sistema.
Foram criadas três classes individuais de transação — BalanceInquiry, Withdrawal e Deposit— para representar as transações que o sistema ATM pode realizar.
A Figura 13.7 mostra os atributos e as operações das classes BalanceInquiry, Withdrawal e Deposit.
Cada uma delas tem um atributo (accountNumber) e uma operação (execute) em comum.
Cada classe requer o atributo accountNumber para especificar a conta a qual a transação se aplica.
Cada classe contém uma operação execute, que a ATM invoca para realizar a transação.
BalanceInquiry, Withdrawal e Deposit representam *tipos de transações*

Utilizando herança para fatorar as características comuns, geramos o seguinte diagrama:

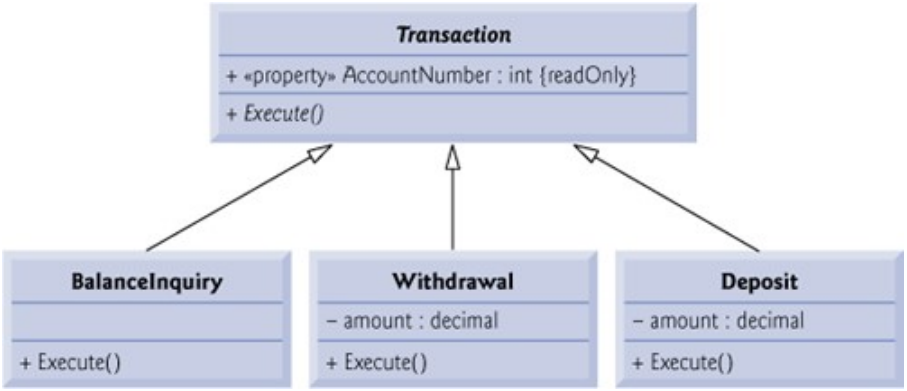


Figura 13.8

O polimorfismo fornece à classe ATM uma maneira elegante de executar todas as transações “no geral”.
A abordagem polimórfica também torna o sistema facilmente extensível.
Para criar um novo tipo de transação, simplesmente crie uma subclasse *Transaction* adicional que sobrescreva o método execute com uma versão do método adequado para a execução de um novo tipo de transação.

A Figura 13.9 apresenta um diagrama de classes atualizado do nosso modelo que incorpora a herança e introduz a classe Transaction.
Modelamos uma associação entre a classe ATM e a classe Transaction para mostrar que a ATM, em um dado momento, executa ou não uma transação (isto é, há zero ou um objeto do tipo Transaction no sistema por vez).
Como uma Withdrawal é um tipo de Transaction, não desenhamos mais uma linha de associação diretamente entre a classe ATM e a classe Withdrawal.
A subclasse Withdrawal herda a associação da superclasse Transaction com a classe ATM.
As subclasses BalanceInquiry e Deposit também herdam essa associação; portanto, as associações anteriormente omitidas entre a ATM e as classes BalanceInquiry e Deposit não existem mais.
Também adicionamos uma associação entre as classes Transaction e BankDatabase (Figura 13.9).
Todas as classes Transaction exigem uma referência a BankDatabase para que possam acessar e modificar as informações da conta.
Mostramos uma associação entre as classes Transaction e Screen.
Todas as classes Transaction exibem a saída para o usuário via classe Screen.

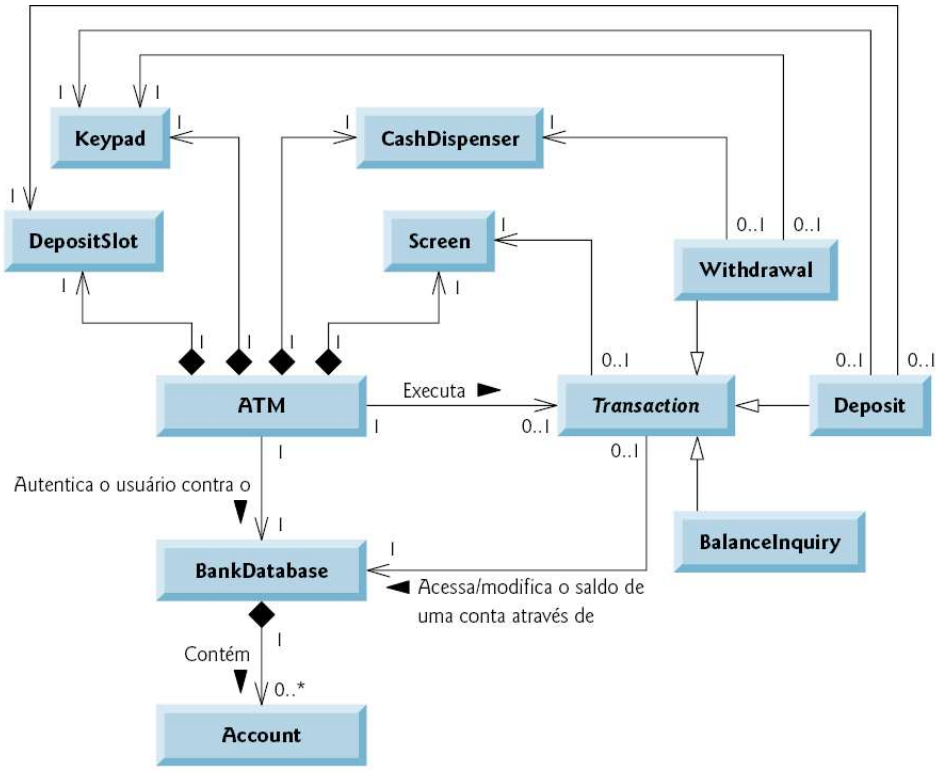
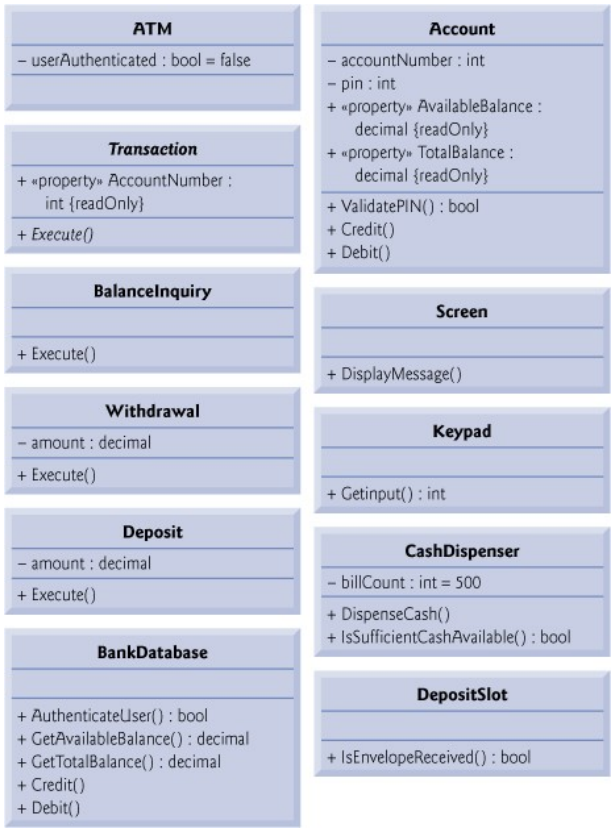


Figura 13.9 | Diagrama de classe do sistema ATM (incorporando a herança). Observe que o nome da classe abstrata `Transaction` aparece em itálico.

As classes depois de incorporada a herança ao sistema:



Pedido 6:

Para todos os pedidos abaixo elabore a documentação a ser gerada.

6.1 Refatoreclasse Withdrawal, implementando a herança acima.

A classe Withdrawal estende Transaction e representa uma transação ATM de consulta de saldo.

Essa classe vai precisar de um objeto Keypad e CashDispenser para realizar as suas operações.

O método execute() precisará de objetos BankDatabase e Screen para realizar suas tarefas.

Precisará de um método displayMenuOfAmounts() para mostrar as opções de saque.

6.2 Implemente a classe Transaction sabendo que:

A classe Transaction é uma superclasse abstrata que representa a noção de uma transação no ATM.

Ela contém os recursos comuns das subclasses BalanceInquiry, Withdrawal e Deposit.

A classe tem três métodos public *get*— getAccountNumber, getScreen, e getBankDatabase.

Eles são herdados pelas subclasses Transaction e utilizados para obter acesso aos atributos private da classe Transaction.

De acordo com o diagrama de classes, essa classe devera conter um objeto Screen onde serão apresentadas as transações e um objeto BankDatabase com informações sobre a conta.

6.3 Implemente as classes BalanceInquiry e Deposit sabendo que:

A classe BalanceInquiry estende Transaction e representa uma transação ATM de consulta de saldo.

BalanceInquiry não tem seus próprios atributos, mas herda os atributos Transaction AccountNumber, screen e bankDatabase, que são acessíveis por meio dos métodos public *get* de Transaction.

A Figura 13.9, acima, modela as associações entre a classe Withdrawal e as classes Keypad e CashDispenser, com as quais as linhas 7–8 implementam os atributos por tipo de referência keypad e cashDispenser, respectivamente.

A classe Deposit vai precisar de objetos Keypad e DepositSlot e de um método promptForDepositAmount(), para que o cliente entre a quantia depositada.

Um diagrama de atividades modela aspectos do comportamento do sistema.

Modela o **fluxo de trabalho** de um objeto durante a execução do programa.

Modela as **ações** que o objeto realizará e em qual ordem.

A UML representa uma ação como um estado de ação modelado por um retângulo com seus lados esquerdo e direito substituídos por arcos convexos.

Cada um contém uma expressão de ação que especifica uma ação a ser realizada.

Uma seta conecta dois estados de ação, indicando a ordem em que ocorrem as ações.

O círculo sólido representa o estado inicial da atividade — o começo do fluxo de trabalho antes de o objeto realizar a ações modeladas.

O círculo sólido dentro de um círculo vazado representa o estado final — o fim do fluxo de trabalho depois de o objeto realizar as ações modeladas.

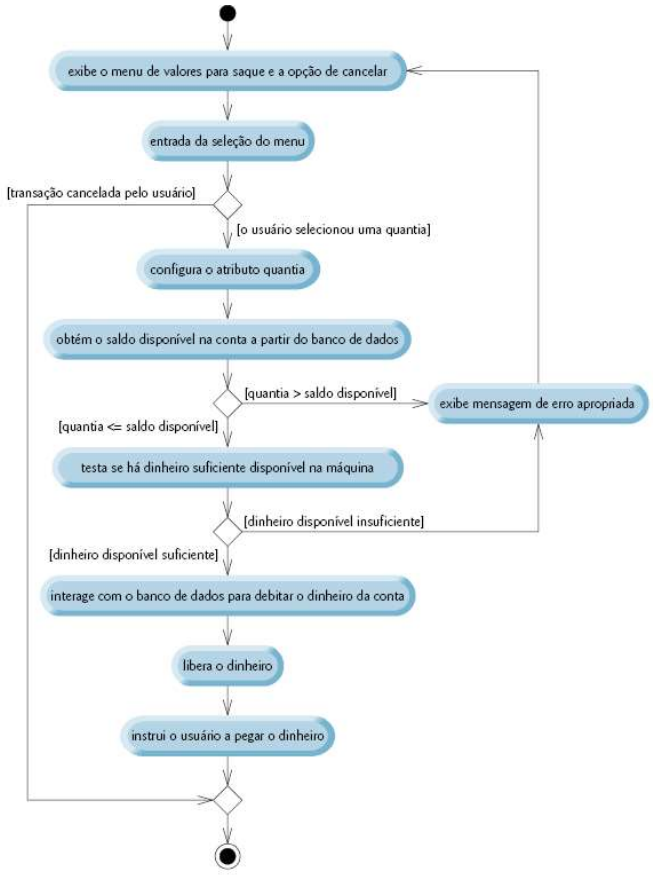


Figura 12.15 | Diagrama de atividades para uma transação de retirada.

Incorporando documentação ao sistema ATM

O Visual Studio oferece um poderoso recurso de documentação que permite descrever propriedades, métodos e seus parâmetros, exibindo isso no *intellisense* ao se tentar acessar um campo documentado.

Para adicionar a descrição a um método, classe ou propriedade, basta posicionar o cursor na linha acima da sua declaração e digitar /// (três barras seguidas). O Visual Studio irá automaticamente completar o código, identificando que ali será inserida a documentação daquele item.

```
5   namespace br.com.Money
6   {
7       /// <summary>
8       /// Class Withdrawal represents an ATM withdrawal transaction
9       /// </summary>
10
        1 referência
11      public class Withdrawal
12      {
13          // attributes
14          private int accountNumber; // account to withdraw funds from
15          private decimal amount; // amount to withdraw
16
```

Tempo para realizar a tarefa: 40 minutos.