J

Código para o estudo de caso do ATM

J.1 Implementação do estudo de caso ATM

Este apêndice contém a implementação funcional completa do sistema ATM que projetamos nas seções "Estudo de caso de engenharia de software" no final dos capítulos 1 a 8 e 10. A implementação abrange 670 linhas de código Java. Consideramos as classes na ordem em que as identificamos na Seção 3.10:

- `ATM`
- `Screen`
- `Keypad`
- `CashDispenser`
- `DepositSlot`
- `Account`
- `BankDatabase`
- `Transaction`
- `BalanceInquiry`
- `Withdrawal`
- `Deposit`

Aplicamos as diretrizes discutidas nas seções 8.19 e 10.9 para codificar essas classes como base na maneira como as modelamos nos diagramas de classes da UML das figuras 10.21 e 10.22. Para desenvolver os corpos para os métodos das classes, utilizamos os diagramas de atividades apresentados na Seção 5.11 e os diagramas de seqüência e de comunicação apresentados na Seção 7.14. Observe que o nosso projeto do ATM não especifica toda a lógica do programa e talvez não especifique todos os atributos e operações necessárias para completar a implementação do ATM. Isso é uma parte normal do processo de um projeto orientado a objetos. À medida que implementamos o sistema, completaremos a lógica do programa e adicionaremos atributos e comportamentos conforme necessário para construir o sistema ATM especificado pelo documento de requisitos na Seção 2.9.

Concluímos a discussão apresentando um aplicativo Java (`ATMCaseStudy`) que inicia o ATM e coloca em uso as outras classes no sistema. Lembre-se de que estamos desenvolvendo uma primeira versão do sistema ATM que é executada em um computador pessoal e utilizamos o teclado e monitor do computador para simular o teclado numérico e a tela do sistema ATM. Também simulamos apenas as ações do dispensador de cédulas e da abertura para depósito. Tentamos, porém, implementar o sistema de modo que versões reais de hardware desses dispositivos possam ser integradas sem alterações significativas no código.

J.2 Classe `ATM`

A classe `ATM` (Figura J.1) representa o ATM como um todo. As linhas 6–12 implementam os atributos dessa classe. Determinamos todos, exceto um, esses atributos por meio dos diagramas de classe da UML das figuras. 10.21 e 10.22. Observe que implementamos o atributo `Boolean userAuthenticated` da UML na Figura 10.22 como um atributo `boolean` em Java (linha 6). A linha 7 declara um atributo como não-localizado no nosso projeto em UML — um atributo `int currentAccountNumber` que monitora o número de conta do atual usuário autenticado. Mais adiante, veremos como a classe usa esse atributo. As linhas 8–12 declaram os atributos dos tipos por referência que correspondem às associações da classe `ATM` modeladas no diagrama de classes da Figura 10.21. Esses atributos permitem que o ATM acesse seus componentes (`Screen`, `Keypad`, `CashDispenser` e `DepositSlot`) e interaja com o banco de dados de informações sobre contas do banco (um objeto `BankDatabase`).

```
1  // ATM.Java
2  // Representa um caixa automático
3
4  public class ATM
5  {
6     private boolean userAuthenticated; // se usuário foi autenticado
7     private int currentAccountNumber; // número atual da conta de usuário
8     private Screen screen; // Tela do ATM
9     private Keypad keypad; // Teclado do ATM
10    private CashDispenser cashDispenser; // dispensador de cédulas do ATM
11    private DepositSlot depositSlot; // Abertura para depósito do ATM
12    private BankDatabase bankDatabase; // banco de dados de informações de contas
13
14    // constantes que correspondem às principais opções de menu
15    private static final int BALANCE_INQUIRY = 1;
16    private static final int WITHDRAWAL = 2;
17    private static final int DEPOSIT = 3;
18    private static final int EXIT = 4;
19
20    // construtor sem argumento de ATM inicializa as variáveis de instância
21    public ATM()
22    {
23       userAuthenticated = false; // usuário não foi autenticado para iniciar
24       currentAccountNumber = 0; // nenhum número atual de conta para iniciar
25       screen = new Screen(); // cria a tela
26       keypad = new Keypad(); // cria o teclado numérico
27       cashDispenser = new CashDispenser(); // cria o dispensador de cédulas
28       depositSlot = new DepositSlot(); // cria a abertura para depósito
29       bankDatabase = new BankDatabase(); // cria o banco de dados de informações de contas
30    } // fim do construtor ATM sem argumento
31
32    // inicia ATM
33    public void run()
34    {
35       // dá boas-vindas e autentica o usuário; realiza transações
36       while ( true )
37       {
38          // faz um loop enquanto o usuário ainda não está autenticado
39          while ( !userAuthenticated )
40          {
41             screen.displayMessageLine( "\nWelcome!" );
42             authenticateUser(); // autentica o usuário
43          } // fim do while
44
45          performTransactions(); // o usuário agora está autenticado
46          userAuthenticated = false; // reinicializa antes da próxima sessão do ATM
47          currentAccountNumber = 0; // reinicializa antes da próxima sessão do ATM
48          screen.displayMessageLine( "\nThank you! Goodbye!" );
49       } // fim do while
50    } // fim do método run
51
52    // tenta autenticar o usuário contra o banco de dados
53    private void authenticateUser()
54    {
55       screen.displayMessage( "\nPlease enter your account number: " );
```

Figura J.1 A classe ATM representa o ATM. (Parte 1 de 3.)

```
56          int accountNumber = keypad.getInput(); // insere o número de conta
57          screen.displayMessage( "\nEnter your PIN: " ); // solicita o PIN
58          int pin = keypad.getInput(); // insere o PIN
59
60          // configura userAuthenticated como um valor booleano retornado pelo banco de dados
61          userAuthenticated =
62             bankDatabase.authenticateUser( accountNumber, pin );
63
64          // verifica se a autenticação foi bem-sucedida
65          if ( userAuthenticated )
66          {
67             currentAccountNumber = accountNumber; // salva a conta do usuário #
68          } // fim do if
69          else
70             screen.displayMessageLine(
71                "Invalid account number or PIN. Please try again." );
72       } // fim do método authenticateUser
73
74       // exibe o menu principal e realiza transações
75       private void performTransactions()
76       {
77          // variável local para armazenar a transação atualmente processada
78          Transaction currentTransaction = null;
79
80          boolean userExited = false; // usuário optou por não sair
81
82          // faz um loop enquanto o usuário não escolher a opção para sair do sistema
83          while ( !userExited )
84          {
85             // mostra o menu principal e obtém a seleção de usuário
86             int mainMenuSelection = displayMainMenu();
87
88             // decide como prosseguir com base na seleção de menu feita pelo usuário
89             switch ( mainMenuSelection )
90             {
91                // o usuário optou por realizar um entre três tipos de transações
92                case BALANCE_INQUIRY:
93                case WITHDRAWAL:
94                case DEPOSIT:
95
96                   // inicializa como o novo objeto do tipo escolhido
97                   currentTransaction =
98                      createTransaction( mainMenuSelection );
99
100                  currentTransaction.execute(); // executa a transação
101                  break;
102               case EXIT: // usuário optou por terminar a sessão
103                  screen.displayMessageLine( "\nExiting the system..." );
104                  userExited = true; // essa sessão de ATM deve terminar
105                  break;
106               default: // usuário não inseriu um inteiro de 1 a 4
107                  screen.displayMessageLine(
108                     "\nYou did not enter a valid selection. Try again." );
109                  break;
110            } // fim de switch
```

Figura J.1 A classe ATM representa o ATM. (Parte 2 de 3.)

```
111       } // fim do while
112    } // fim do método performTransactions
113
114    // exibe o menu principal e retorna uma seleção de entrada
115    private int displayMainMenu()
116    {
117       screen.displayMessageLine( "\nMain menu:" );
118       screen.displayMessageLine( "1 - View my balance" );
119       screen.displayMessageLine( "2 - Withdraw cash" );
120       screen.displayMessageLine( "3 - Deposit funds" );
121       screen.displayMessageLine( "4 - Exit\n" );
122       screen.displayMessage( "Enter a choice: " );
123       return keypad.getInput(); // retorna a seleção do usuário
124    } // fim do método displayMainMenu
125
126    // retorna o objeto da subclasse de Transaction especificada
127    private Transaction createTransaction( int type )
128    {
129       Transaction temp = null; // variável Transaction temporária
130
131       // determina qual tipo de Transaction criar
132       switch ( type )
133       {
134          case BALANCE_INQUIRY: // cria uma nova transação BalanceInquiry
135             temp = new BalanceInquiry(
136                currentAccountNumber, screen, bankDatabase );
137             break;
138          case WITHDRAWAL: // cria uma nova transação Withdrawal
139             temp = new Withdrawal( currentAccountNumber, screen,
140                bankDatabase, keypad, cashDispenser );
141             break;
142          case DEPOSIT: // cria uma nova transação Deposit
143             temp = new Deposit( currentAccountNumber, screen,
144                bankDatabase, keypad, depositSlot );
145             break;
146       } // fim de switch
147
148       return temp; // retorna o objeto recém-criado
149    } // fim do método createTransaction
150 } // fim da classe ATM
```

Figura J.1 A classe ATM representa o ATM. (Parte 3 de 3.)

As linhas 15–18 declaram constantes inteiras que correspondem às quatro opções no menu principal do ATM (consulta de saldos, retirada, depósito e saída). As linhas 21–30 declaram o construtor da classe `ATM`, que inicializa os atributos da classe. Quando um objeto `ATM` é criado pela primeira vez, nenhum usuário é autenticado, assim a linha 23 inicializa `userAuthenticated` como `false`. Da mesma forma, a linha 24 inicializa `currentAccountNumber` como `0` porque ainda não há nenhum usuário atual. As linhas 25–28 instanciam novos objetos para representar as partes do ATM. Lembre-se de que a classe `ATM` tem relacionamentos de composição com as classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`, portanto a classe `ATM` é responsável pela criação dessas classes. A linha 29 cria um `BankDatabase`. [*Nota*: Se esse sistema ATM fosse real, a classe `ATM` receberia uma referência a um objeto existente de banco de dados criado pelo banco. Entretanto, nessa implementação só estamos simulando o banco de dados do banco, portanto a classe `ATM` cria o objeto `BankDatabase` com o qual interage.]

O diagrama de classes da Figura 10.22 não lista nenhuma operação para a classe `ATM`. Agora, implementaremos uma operação (o método `public`) na classe `ATM` que permite a um cliente externo da classe (a classe `ATMCaseStudy`) instruir o `ATM` a executar. O método `run` de `ATM` (linhas 33–50) utiliza um loop infinito (linhas 36–49) para repetidamente dar boas-vindas a um usuário, tenta autenticar o usuário e, se a autenticação for bem-sucedida, permite que o usuário realize as transações. Depois que um usuário autenticado realiza as transações desejadas e escolhe sair, o ATM redefine a si mesmo, exibe uma mensagem de adeus para o usuário e reinicia o processo. Aqui,

utilizamos um loop infinito para simular o fato de que um ATM parece executar continuamente até o banco desligá-lo (uma ação além do controle do usuário). Um usuário do ATM tem a opção de sair do sistema, mas não a capacidade de desligar o ATM completamente.

O loop infinito do método interno `run`, linhas 39–43 fazem com que o ATM repetidamente emita uma mensagem de bem-vindo e tenta autenticar o usuário contanto que o usuário não esteja autenticado (ou seja, `!userAuthenticated` é `true`). A linha 41 invoca o método `displayMessageLine` de `screen` da classe `ATM` para exibir uma mensagem de bem-vindo. Como ocorre com o método `displayMessage` de `Screen` projetado no estudo de caso, o método `displayMessageLine` (declarado nas linhas 13–16 da Figura J.2) exibe uma mensagem para o usuário, mas esse método também gera a saída de uma nova linha depois de exibir a mensagem. Adicionamos esse método durante a implementação para fornecer aos clientes da classe `Screen` mais controle sobre o posicionamento das mensagens exibidas. A linha 42 invoca o método utilitário `private` `authenticateUser` da classe `ATM` (declarado nas linhas 53–72) para tentar autenticar o usuário.

Referimos ao documento de requisitos para determinar os passos necessários a fim de autenticar o usuário antes de permitir que as transações ocorram. A linha 55 do método `authenticateUser` invoca o método `displayMessage` de `screen` da classe `ATM` para solicitar que o usuário insira um número de conta. A linha 56 invoca o método `getInput` de `keypad` da classe `ATM` para obter a entrada do usuário e, então, armazena o valor inteiro inserido pelo usuário em uma variável local `accountNumber`. Em seguida, o método `authenticateUser` solicita que o usuário insira um PIN (linha 57) e armazena o PIN inserido pelo usuário em uma variável local `pin` (linha 58). As linhas 61–62 tentam então autenticar o usuário passando o `accountNumber` e o `pin` inserido pelo usuário para o método `authenticateUser` de `bankDatabase`. A classe `ATM` configura seu atributo `userAuthenticated` como o valor `boolean` retornado por esse método — `userAuthenticated` torna-se `true` se a autenticação for bem-sucedida (se `accountNumber` e `pin` corresponderem àqueles de uma `Account` existente em `bankDatabase`) e, do contrário, permanece `false`. Se `userAuthenticated` for `true`, a linha 67 salvará o número de conta inserido pelo usuário (`accountNumber`) no atributo `currentAccountNumber` de ATM. Os outros métodos da classe `ATM` utilizam essa variável sempre que uma sessão de ATM exige acesso ao número da conta do usuário. Se `userAuthenticated` é `false`, a linhas 70–71 utilizam o método `displayMessageLine` de `screen` para indicar que um número inválido de conta e/ou PIN foi inserido e o usuário deve tentar novamente. Observe que configuramos `currentAccountNumber` somente depois de autenticar o número da conta do usuário e o PIN associado — se o banco de dados não puder autenticar o usuário, `currentAccountNumber` permanecerá `0`.

Depois de o método `run` tentar autenticar o usuário (linha 42), se `userAuthenticated` ainda for `false`, o loop `while` nas linhas 39–43 será executado novamente. Se `userAuthenticated` for `true`, o loop terminará e o controle continuará com a linha 45, que chamará o método utilitário `performTransactions` da classe `ATM`.

O método `performTransactions` (linhas 75–112) executa uma sessão de ATM para um usuário autenticado. A linha 78 declara uma variável `Transaction` local à qual atribuímos um objeto `BalanceInquiry`, `Withdrawal` ou `Deposit` que representa a transação do ATM atualmente em processamento. Observe que aqui utilizamos uma variável `Transaction` para permitir que tiremos proveito do polimorfismo. Também observe que nomeamos essa variável depois do nome de papel incluído no diagrama de classes da Figura 3.21 — `currentTransaction`. A linha 80 declara uma outra variável local — uma `boolean` chamada `userExited` que monitora se o usuário optou por sair. Essa variável controla um loop `while` (linhas 83–111) que permite ao usuário executar um número ilimitado de transações antes de optar por sair. Dentro desse loop, a linha 86 exibe o menu principal e obtém a seleção de menu do usuário chamando o método utilitário `displayMainMenu` de `ATM` (declarado nas linhas 115–124). Esse método exibe o menu principal invocando os métodos de `screen` da classe `ATM` e retorna uma seleção de menu obtido do usuário pela `keypad` da `ATM`. A linha 86 armazena a seleção do usuário retornado por `displayMainMenu` na variável local `mainMenuSelection`.

Depois de obter uma seleção no menu principal, o método `performTransactions` utiliza uma instrução `switch` (linhas 89–110) para responder à seleção apropriadamente. Se `mainMenuSelection` for igual a uma das três constantes inteiras que representa tipos de transação (se o usuário optou por realizar uma transação), as linhas 97–98 chamarão o método utilitário `createTransaction` (declarado nas linhas 127–149) para retornar um objeto recém-instanciado do tipo que corresponde à transação selecionada. A variável `currentTransaction` recebe a referência retornada por `createTransaction`, e a linha 100 invoca o método `execute` dessa transação para executá-lo. Discutiremos o método `execute` de `Transaction` e as três subclasses `Transaction` mais adiante. Observe que atribuímos à variável `Transaction` de `currentTransaction` um objeto de uma das três subclasses `Transaction` para ser possível executar as transações polimorficamente. Por exemplo, se o usuário decidir realizar uma consulta de saldos, `mainMenuSelection` será igual a `BALANCE_INQUIRY`, levando `createTransaction` a retornar um objeto `BalanceInquiry`. Portanto, `currentTransaction` se refere a um `BalanceInquiry`, e invocar `currentTransaction.execute()` resulta em chamar a versão de `BalanceInquiry` do método `execute`.

O método `createTransaction` (linhas 127–149) utiliza uma instrução `switch` (linhas 132–146) para instanciar um novo objeto da subclasse `Transaction` do tipo indicado pelo parâmetro `type`. Lembre-se de que o método `performTransactions` passa `mainMenuSelection` para esse método somente se `mainMenuSelection` contiver um valor correspondente a um dos três tipos de transação. Portanto `type` é igual a `BALANCE_INQUIRY`, `WITHDRAWAL` ou `DEPOSIT`. Cada `case` na instrução `switch` instancia um novo objeto chamando o construtor da subclasse `Transaction` apropriado. Observe que cada construtor contém uma lista única de parâmetros, baseada nos dados específicos necessários para inicializar o objeto da subclasse. Uma `BalanceInquiry` exige apenas o número de conta do usuário atual e referências a `bankDatabase` e `screen` da classe `ATM`. Além desses parâmetros, um `Withdrawal` requer referências a `keypad` e `cashDispenser` da classe `ATM` e uma `Deposit` requer referências a `keypad` e `depositSlot` da classe `ATM`. Discutimos as classes de transação em mais detalhes nas Seções J.9 a J.12.

Depois de executar uma transação (linha 100 em `performTransactions`), `userExited` permanece `false`, e o loop `while` nas linhas 83–111 é repetido, retornando o usuário ao menu principal. Entretanto, se um usuário não realizar uma transação e, em vez disso, selecionar uma opção de saída no menu principal, a linha 104 configura `userExited` como `true` fazendo com que a condição do loop

while (!userExited) torne-se false. Esse while é a instrução final do método performTransactions, portanto o controle retorna ao método chamador run. Se o usuário inserir uma seleção inválida no menu principal (não um inteiro entre 1 e 4), as linhas 107–108 exibirão uma mensagem de erro apropriada, userExited permanecerá false e o usuário retornará ao menu principal para tentar novamente.

Quando o controle de performTransactions retorna ao método run, o usuário optou por sair do sistema e as linhas 46–47 redefinem os atributos userAuthenticated e currentAccountNumber da classe ATM para preparar-se para o próximo usuário do ATM. A linha 48 exibe uma mensagem de adeus antes de o ATM recomeçar e dar boas-vindas ao próximo usuário.

J.3 Classe Screen

A classe Screen (Figura J.2) representa a tela do ATM e encapsula todos os aspectos da exibição de saída para o usuário. A classe Screen lembra a tela de um ATM real de um monitor de computador e gera a saída de mensagens de texto utilizando os métodos-padrão de saída de console System.out.print, System.out.println e System.out.printf. Nesse estudo de caso, projetamos a classe Screen com uma operação — displayMessage. Para maior flexibilidade na exibição de mensagens na Screen, agora declaramos três métodos de Screen — displayMessage, displayMessageLine e displayDollarAmount.

```
1  // Screen.java
2  // Representa a tela do ATM
3
4  public class Screen
5  {
6     // exibe uma mensagem sem retorno de carro
7     public void displayMessage( String message )
8     {
9        System.out.print( message );
10    } // fim do método displayMessage
11
12    // exibe uma mensagem com um retorno de carro
13    public void displayMessageLine( String message )
14    {
15       System.out.println( message );
16    } // fim do método displayMessageLine
17
18    // exibe um valor em dólares
19    public void displayDollarAmount( double amount )
20    {
21       System.out.printf( "$%,.2f", amount );
22    } // fim do método displayDollarAmount
23 } // fim da classe Screen
```

Figura J.2 A classe Screen representa a tela do ATM.

O método displayMessage (linhas 7–10) recebe uma String como um argumento e a imprime no console utilizando System.out.print. O cursor permanece na mesma linha, tornando esse método apropriado para exibir solicitações para o usuário. O método displayMessageLine (linhas 13–16) faz a mesma coisa por meio de System.out.println, que gera a saída de uma linha para mover o cursor para a próxima linha. Por fim, o método displayDollarAmount (linhas 19–22) gera a saída de uma quantia em dólares adequadamente formatada (por exemplo, $1,234.56). A linha 21 utiliza o método System.out.printf para gerar a saída de um valor de double formatado com vírgulas a fim de aumentar a legibilidade, e duas casas decimais. Consulte o Capítulo 28 para informações adicionais sobre como formatar a saída com printf.

J.4 Classe Keypad

A classe Keypad (Figura J.3) representa o teclado numérico do ATM e é responsável por receber todas as entradas de usuário. Lembre-se de que estamos simulando esse hardware, portanto utilizamos o teclado do computador para simular o teclado numérico. Utilizamos a classe Scanner para obter a entrada de console do usuário. O teclado de um computador contém muitas teclas não encontradas no teclado numérico do ATM. Entretanto, supomos que o usuário pressione somente as teclas presentes no teclado de um computador que também aparecem no teclado numérico — as teclas numeradas de 0–9 e a tecla *Enter*.

A linha 3 da classe Keypad utiliza import para importar a classe Scanner para uso na classe Keypad. A linha 7 declara a variável input de Scanner como uma variável de instância. A linha 12 no construtor cria um objeto Scanner que lê a entrada a partir do fluxo de

entrada padrão (`System.in`) e atribui a referência do objeto à variável `input`. O método `getInput` (declarado nas linhas 16–19) invoca o método `nextInt` de `Scanner` (linha 18) para retornar o próximo inteiro inserido pelo usuário. [*Nota*: O método `nextInt` pode lançar uma `InputMismatchException` se o usuário inserir um não-inteiro na entrada. Uma vez que o teclado numérico de um ATM real só permite entrada de inteiros, supomos que nenhuma exceção ocorrerá e não tentaremos corrigir esse problema. Consulte o Capítulo 13 para informações sobre como capturar exceções.] Lembre-se de que `nextInt` obtém todas as entradas utilizadas pelo ATM. O método `getInput` de `Keypad` simplesmente retorna o inteiro inserido pelo usuário. Se um cliente da classe `Keypad` exigir uma entrada que deve satisfazer alguns critérios em particular (um número que corresponda a uma opção válida no menu), o cliente deverá realizar a verificação de erros apropriada.

```
1  // Keypad.java
2  // Representa o teclado do ATM
3  import java.util.Scanner; // o programa utiliza Scanner para obter a entrada do usuário
4
5  public class Keypad
6  {
7     private Scanner input; // lê os dados na linha de comando
8
9     // o construtor sem argumento inicializa a classe Scanner
10    public Keypad()
11    {
12       input = new Scanner( System.in );
13    } // fim do construtor Keypad sem argumentos
14
15    // retorna um valor inteiro inserido pelo usuário
16    public int getInput()
17    {
18       return input.nextInt(); // supomos que o usuário insira um inteiro
19    } // fim do método getInput
20 } // fim da classe Keypad
```

Figura J.3 A classe `Keypad` representa o teclado do ATM.

J.5 Classe `CashDispenser`

A classe `CashDispenser` (Figura J.4) representa o dispensador de cédulas do ATM. A linha 7 declara a constante `INITIAL_COUNT`, que indica a contagem inicial de cédulas no dispensador de cédulas quando o ATM é inicializado (isto é, `500`). A linha 8 implementa o atributo `count` (modelado na Figura 10.22), que monitora o número de cédulas que permanece no `CashDispenser` em um dado momento. O construtor (linhas 11–14) configura `count` como a contagem inicial. A classe `CashDispenser` contém dois métodos `public` — `dispenseCash` (linhas 17–21) e `isSufficientCashAvailable` (linhas 24–32). A classe confia no fato de que um cliente (`Withdrawal`) chama `dispenseCash` somente depois de estabelecer que há cédulas suficientes disponíveis chamando `isSufficientCashAvailable`. Portanto, `dispenseCash` apenas simula o ato de entregar a quantia solicitada sem verificar se há cédulas suficientes disponíveis.

O método `isSufficientCashAvailable` (linhas 24–32) contém um parâmetro `amount` que especifica a quantia de cédulas em questão. A linha 26 calcula o número de cédulas de US$ 20 requeridas para entregar o `amount` especificado. O ATM permite que o usuário escolha somente quantias de retirada que sejam múltiplos de US$ 20, assim dividimos `amount` por `20` para obter o número de `billsRequired`. As linhas 28–31 retornam `true` se o `count` de `CashDispenser` for maior ou igual a `billsRequired` (isto é, há cédulas suficientes disponíveis) e `false`, caso contrário (isto é, não há cédulas suficientes). Por exemplo, se um usuário deseja sacar US$ 80 (`billsRequired` é `4`), mas só há três cédulas (`count` é `3`), o método retorna `false`.

```
1  // CashDispenser.java
2  // Representa o dispensador de cédulas do ATM
3
4  public class CashDispenser
5  {
6     // o número inicial padrão de cédulas no dispensador de cédulas
7     private final static int INITIAL_COUNT = 500;
8     private int count; // número de cédulas de US$ 20 remanescente
9
```

Figura J.4 A classe `CashDispenser` representa o dispensador de cédulas do ATM. (Parte 1 de 2.)

```
10    // construtor sem argumento CashDispenser inicializa a count para o padrão
11    public CashDispenser()
12    {
13       count = INITIAL_COUNT; // configura atributo count como o padrão
14    } // fim do construtor CashDispenser
15
16    // simula a entrega da quantia especificada de cédulas
17    public void dispenseCash( int amount )
18    {
19       int billsRequired = amount / 20; // número de cédulas de US$ 20 requerido
20       count -= billsRequired; // atualiza a contagem das cédulas
21    } // fim do método dispenseCash
22
23    // indica se o dispensador de cédulas pode entregar a quantia desejada
24    public boolean isSufficientCashAvailable( int amount )
25    {
26       int billsRequired = amount / 20; // número de cédulas de US$ 20 requerido
27
28       if ( count >= billsRequired  )
29          return true; // há cédulas suficientes disponíveis
30       else
31          return false; // não há cédulas suficientes disponíveis
32    } // fim do método isSufficientCashAvailable
33 } // fim da classe CashDispenser
```

Figura J.4 A classe `CashDispenser` representa o dispensador de cédulas do ATM. (Parte 2 de 2.)

O método `dispenseCash` (linhas 17–21) simula a entrega de cédulas. Se nosso sistema estivesse acoplado a um dispensador de cédulas de um hardware real, esse método interagiria com o dispositivo de hardware para fisicamente entregar cédulas. Nossa versão simulada do método simplesmente diminui a `count` das cédulas remanescentes de acordo com o número requerido para entregar o `amount` especificado (linha 20). Observe que é responsabilidade do cliente da classe (`Withdrawal`) informar o usuário de que cédulas foram entregues — `CashDispenser` não pode interagir diretamente com `Screen`.

J.6 Classe `DepositSlot`

A classe `DepositSlot` (Figura J.5) representa a abertura para depósito do ATM. Como ocorre com a versão da classe `CashDispenser` apresentada aqui, essa versão da classe `DepositSlot` simplesmente simula a funcionalidade de um hardware real da abertura para depósito. A classe `DepositSlot` não tem atributos e apenas um método — `isEnvelopeReceived` (linhas 8–11) — que indica se um envelope de depósito foi recebido.

```
1  // DepositSlot.java
2  // Representa a abertura para depósito do ATM
3
4  public class DepositSlot
5  {
6     // indica se o envelope foi recebido (sempre retorna true,
7     // porque isso só é uma simulação do software de uma abertura para depósito real)
8     public boolean isEnvelopeReceived()
9     {
10       return true; // o envelope de depósito foi recebido
11    } // fim do método isEnvelopeReceived
12 } // fim da classe DepositSlot
```

Figura J.5 A classe `DepositSlot` representa a abertura para depósito do ATM.

Lembre-se de que no documento de requisitos o ATM permite que o usuário insira um envelope dentro de no máximo dois minutos. A versão atual do método `isEnvelopeReceived` simplesmente retorna `true` imediatamente (linha 10), pois isso só é uma simulação de software e supomos que o usuário inseriu um envelope dentro do prazo exigido. Se um hardware real de abertura para depósito estivesse conectado ao nosso sistema, o método `isEnvelopeReceived` poderia ser implementado para esperar no máximo dois minutos a fim de

receber um sinal do hardware de abertura para depósito indicando que o usuário de fato inseriu um envelope de depósito. Se `isEnvelopeReceived` fosse receber esse sinal dentro de dois minutos, o método retornaria `true`. Se dois minutos tivessem passado e o método ainda não tivesse recebido um sinal, o método então retornaria `false`.

J.7 Classe Account

A classe `Account` (Figura J.6) representa uma conta bancária. Cada `Account` tem quatro atributos (modelados na Figura 10.22) — `accountNumber`, `pin`, `availableBalance` e `totalBalance`. As linhas 6–9 implementam esses atributos como campos `private`. A variável `availableBalance` representa a quantia de fundos disponível para saque. A variável `totalBalance` representa a quantia de fundos disponível, mais a quantia de fundos depositados ainda aguardando confirmação ou compensação.

A classe `Account` contém um construtor (linhas 12–19) que recebe um número de conta, o PIN estabelecido para a conta, o saldo inicial disponível e o saldo inicial total como argumentos. As linhas 15–18 atribuem esses valores aos atributos da classe (campos).

O método `validatePIN` (linhas 22–28) determina se um PIN especificado pelo usuário (o parâmetro `userPIN`) corresponde ao PIN associado com a conta (o atributo `pin`). Lembre-se de que modelamos esse parâmetro do método `userPIN` no diagrama de classes da UML da Figura 6.23. Se os dois PINs corresponderem, o método retornará `true` (linha 25); caso contrário, retornará `false` (linha 27).

Os métodos `getAvailableBalance` (linhas 31–34) e `getTotalBalance` (linhas 37–40) são métodos *get* que retornam os valores dos atributos `availableBalance` e `totalBalance` de `double`, respectivamente.

O método `credit` (linhas 43–46) adiciona uma quantia de dinheiro (o parâmetro `amount`) a uma `Account` como parte de uma transação de depósito. Observe que esse método adiciona o `amount` somente para atribuir `totalBalance` (linha 45). O dinheiro creditado a uma conta durante um depósito não se torna disponível imediatamente, assim modificamos somente o saldo total. Supomos que o banco atualize o saldo disponível apropriadamente em um momento posterior. Nossa implementação da classe `Account` inclui somente os métodos necessários para executar as transações no ATM. Portanto, omitimos os métodos que alguns outros sistemas bancários invocariam para adicionar o atributo `availableBalance` (para confirmar um depósito) ou subtrair do atributo `totalBalance` (para rejeitar um depósito).

```
1  // Account.java
2  // Representa uma conta bancária
3
4  public class Account
5  {
6     private int accountNumber; // número da conta
7     private int pin; // PIN para autenticação
8     private double availableBalance; // fundos disponíveis para saque
9     private double totalBalance; // fundos disponíveis + depósitos pendentes
10
11    // O construtor Account inicializa os atributos
12    public Account( int theAccountNumber, int thePIN,
13       double theAvailableBalance, double theTotalBalance )
14    {
15       accountNumber = theAccountNumber;
16       pin = thePIN;
17       availableBalance = theAvailableBalance;
18       totalBalance = theTotalBalance;
19    } // fim do construtor Account
20
21    // determina se um PIN especificado pelo usuário corresponde ao PIN em Account
22    public boolean validatePIN( int userPIN )
23    {
24       if ( userPIN == pin )
25          return true;
26       else
27          return false;
28    } // fim do método validatePIN
29
30    // retorna o saldo disponível
31    public double getAvailableBalance()
32    {
```

Figura J.6 A classe `Account` representa uma conta bancária. (Parte 1 de 2.)

```
33          return availableBalance;
34       } // fim de getAvailableBalance
35
36       // retorna o saldo total
37       public double getTotalBalance()
38       {
39          return totalBalance;
40       } // fim do método getTotalBalance
41
42       // credita uma quantia à conta
43       public void credit( double amount )
44       {
45          totalBalance += amount; // adiciona ao saldo total
46       } // fim do método credit
47
48       // debita uma quantia da conta
49       public void debit( double amount )
50       {
51          availableBalance -= amount; // subtrai do saldo disponível
52          totalBalance -= amount; // subtrai do saldo total
53       } // fim do método debit
54
55       // retorna o número da conta
56       public int getAccountNumber()
57       {
58          return accountNumber;
59       } // fim do método getAccountNumber
60    } // fim da classe Account
```

Figura J.6 A classe `Account` representa uma conta bancária. (Parte 2 de 2.)

O método `debit` (linhas 49–53) subtrai uma quantia de dinheiro (o parâmetro `amount`) de uma `Account` como parte de uma transação de saque. Esse método subtrai o `amount` dos dois atributos `availableBalance` (linha 51) e `totalBalance` (linha 52), porque um saque afeta ambas as medidas do saldo de uma conta.

O método `getAccountNumber` (linhas 56–59) fornece acesso a `accountNumber` de uma `Account`. Incluímos esse método na nossa implementação de modo que um cliente da classe (`BankDatabase`) possa identificar uma `Account` particular. Por exemplo, `BankDatabase` contém muitos objetos `Account` e pode invocar esse método em cada um dos seus objetos `Account` para localizar aquele com um número específico de conta.

J.8 Classe BankDatabase

A classe `BankDatabase` (Figura J.7) modela o banco de dados do banco com o qual o ATM interage para acessar e modificar informações da conta de um usuário. Determinamos um atributo do tipo por referência para a classe `BankDatabase` com base no seu relacionamento de composição com a classe `Account`. Lembre-se de que um `BankDatabase` é composto de zero ou mais objetos da classe `Account` (Figura 10.21). A linha 6 implementa o atributo `accounts` — um array de objetos `Account` — para implementar esse relacionamento de composição. A classe `BankDatabase` tem um construtor sem argumento (linhas 9–14) que inicializa `accounts` para que elas contenham um conjunto de novos objetos `Account`. Em consideração ao teste do sistema, declaramos `accounts` para conter somente dois elementos no array (linha 11), que instanciamos como os novos objetos `Account` com os dados do teste (linhas 12–13). Observe que o construtor `Account` tem quatro parâmetros — o número da conta e o PIN atribuídos à conta, o saldo inicial disponível e o saldo inicial total.

Lembre-se de que a classe `BankDatabase` serve como um intermediário entre a classe `ATM` e os objetos `Account` reais que contêm informações sobre a conta de um usuário. Portanto, os métodos da classe `BankDatabase` não fazem nada além de invocar os métodos correspondentes do objeto `Account` pertencente ao atual usuário do ATM.

Incluímos o método utilitário `private` de `getAccount` (linhas 17–28) para permitir que a `BankDatabase` obtenha uma referência a um `Account` particular dentro do array de `accounts`. Para localizar a `Account` do usuário, a `BankDatabase` compara o valor retornado pelo método `getAccountNumber` de cada elemento de `accounts` com um número especificado de conta até encontrar uma correspondência. As linhas 20–25 percorrem o array de `accounts`. Se o número de conta da `currentAccount` for igual ao valor do parâmetro `accountNumber`, o método retornará imediatamente a `currentAccount`. Se nenhuma conta tiver o número de conta dado, a linha 27 retornará `null`.

```
1  // BankDatabase.java
2  // Representa o banco de dados de informações de contas bancárias
3
4  public class BankDatabase
5  {
6     private Account accounts[]; // array de Accounts
7
8     // construtor BankDatabase sem argumento inicializa as contas
9     public BankDatabase()
10    {
11       accounts = new Account[ 2 ]; // apenas 2 contar para teste
12       accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
13       accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );
14    } // fim do construtor BankDatabase sem argumento
15
16    // recupera o objeto Account que contém o número de conta especificado
17    private Account getAccount( int accountNumber )
18    {
19       // faz um loop pelas contas procurando uma correspondência com o número de conta
20       for ( Account currentAccount : accounts )
21       {
22          // retorna a conta atual se uma correspondência for localizada
23          if ( currentAccount.getAccountNumber() == accountNumber )
24             return currentAccount;
25       } // fim do for
26
27       return null; // se nenhuma correspondência com uma conta foi localizada, retorna null
28    } // fim do método getAccount
29
30    // determina se número da conta e PIN especificados pelo usuário correspondem
31    // àqueles de uma conta no banco de dados
32    public boolean authenticateUser( int userAccountNumber, int userPIN )
33    {
34       // tenta recuperar a conta com o número da conta
35       Account userAccount = getAccount( userAccountNumber );
36
37       // se a conta existir, retorna o resultado do método validatePIN de Account
38       if ( userAccount != null )
39          return userAccount.validatePIN( userPIN );
40       else
41          return false; // número de conta não foi localizado, portanto retorna false
42    } // fim do método authenticateUser
43
44    // retorna o saldo disponível de Account com o número da conta especificado
45    public double getAvailableBalance( int userAccountNumber )
46    {
47       return getAccount( userAccountNumber ).getAvailableBalance();
48    } // fim do método getAvailableBalance
49
50    // retorna o saldo total de Account com o número da conta especificado
51    public double getTotalBalance( int userAccountNumber )
52    {
53       return getAccount( userAccountNumber ).getTotalBalance();
54    } // fim do método getTotalBalance
55
```

Figura J.7 A classe `BankDatabase` representa o banco de dados de informações de contas do banco. (Parte 1 de 2.)

```
56    // credita uma quantia a Account com o número da conta especificado
57    public void credit( int userAccountNumber, double amount )
58    {
59       getAccount( userAccountNumber ).credit( amount );
60    } // fim do método credit
61
62    // debita uma quantia da Account com número da conta especificado
63    public void debit( int userAccountNumber, double amount )
64    {
65       getAccount( userAccountNumber ).debit( amount );
66    } // fim do método debit
67 } // fim da classe BankDatabase
```

Figura J.7 A classe `BankDatabase` representa o banco de dados de informações de contas do banco. (Parte 2 de 2.)

O método `authenticateUser` (linhas 32–42) aprova ou desaprova a identidade de um usuário do ATM. Esse método recebe o número e o PIN da conta especificados pelo usuário como argumentos e indica se eles correspondem ao número da conta e PIN de uma `Account` no banco de dados. A linha 35 chama o método `getAccount`, que retorna uma `Account` com `userAccountNumber` como seu número de conta ou `null` para indicar que `userAccountNumber` é inválido. Se `getAccount` retornar um objeto `Account`, a linha 39 retornará o valor `boolean` retornado pelo método `validatePIN` desse objeto. Observe que o método `authenticateUser` de `BankDatabase` não realiza a comparação do PIN por si só — em vez disso, ele encaminha `userPIN` para que o método `validatePIN` do objeto `Account` faça isso. O valor retornado pelo método `validatePIN` de `Account` indica se o PIN especificado pelo usuário corresponde ao PIN da `Account` do usuário, portanto o método `authenticateUser` simplesmente retorna esse valor ao cliente da classe (isto é, `ATM`).

O método `BankDatabase` confia na classe `ATM` para invocar o método `authenticateUser` e receber um valor de retorno de `true` antes de permitir que o usuário realize as transações. `BankDatabase` também confia no fato de que cada objeto `Transaction` criado pelo `ATM` contém o número de conta válido do atual usuário autenticado e que esse é o número de conta passado para os métodos `BankDatabase` remanescentes como o argumento `userAccountNumber`. Os métodos `getAvailableBalance` (linhas 45–48), `getTotalBalance` (linhas 51–54), `credit` (linhas 57–60) e `debit` (linhas 63–66), portanto, apenas recuperam o objeto `Account` do usuário com o método utilitário `getAccount` e então invocam o método `Account` apropriado nesse objeto. Sabemos que as chamadas a `getAccount` dentro desses métodos nunca retornarão `null`, porque `userAccountNumber` deve fazer referência a uma `Account` existente. Observe que `getAvailableBalance` e `getTotalBalance` retornam os valores retornados pelos métodos `Account` correspondentes. Também observe que `credit` e `debit` simplesmente redirecionam o parâmetro `amount` para os métodos `Account` que eles invocam.

```
1  // Transaction.java
2  // A superclasse abstrata Transaction representa uma transação no ATM
3
4  public abstract class Transaction
5  {
6     private int accountNumber; // indica conta envolvida
7     private Screen screen; // Tela do ATM
8     private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
9
10    // Construtor de Transaction invocado pelas subclasses utilizando super()
11    public Transaction( int userAccountNumber, Screen atmScreen,
12       BankDatabase atmBankDatabase )
13    {
14       accountNumber = userAccountNumber;
15       screen = atmScreen;
16       bankDatabase = atmBankDatabase;
17    } // fim do construtor de Transaction
18
19    // retorna o número da conta
20    public int getAccountNumber()
21    {
22       return accountNumber;
23    } // fim do método getAccountNumber
```

Figura J.8 A superclasse abstrata `Transaction` representa uma transação no ATM. (Parte 1 de 2.)

```
24
25    // retorna a referência à tela
26    public Screen getScreen()
27    {
28       return screen;
29    } // fim do método getScreen
30
31    // retorno a referência ao banco de dados da instituição financeira
32    public BankDatabase getBankDatabase()
33    {
34       return bankDatabase;
35    } // fim do método getBankDatabase
36
37    // realiza a transação (sobrescrita por cada subclasse)
38    abstract public void execute();
39 } // fim da classe Transaction
```

Figura J.8 A superclasse abstrata `Transaction` representa uma transação no ATM. (Parte 2 de 2.)

J.9 Classe `Transaction`

A classe `Transaction` (Figura J.8) é uma superclasse abstrata que representa o conceito de uma transação no ATM. Ela contém os recursos comuns das subclasses `BalanceInquiry`, `Withdrawal` e `Deposit`. Essa classe expande o código 'esqueleto' desenvolvido inicialmente na Seção 10.9. A linha 4 declara essa classe como `abstract`. As linhas 6–8 declaram os atributos `private` da classe. Lembre-se, no diagrama de classes da Figura 10.22, de que a classe `Transaction` contém um atributo `accountNumber` (linha 6) que indica a conta envolvida na `Transaction`. Derivamos os atributos `screen` (linha 7) e `bankDatabase` (linha 8) das associações da classe `Transaction` modeladas na Figura 10.21 — todas as transações requerem acesso à tela do ATM e ao banco de dados da instituição financeira.

A classe `Transaction` tem um construtor (linhas 11–17) que recebe o número atual da conta do usuário que se refere à tela do ATM e aos dados do banco como os argumentos. Como `Transaction` é uma classe abstrata, esse construtor nunca será chamado diretamente para instanciar os objetos de `Transaction`. Em vez disso, os construtores das subclasses `Transaction` utilizarão `super` para invocar esse construtor.

A classe `Transaction` tem três métodos *get* `public` — `getAccountNumber` (linhas 20–23), `getScreen` (linhas 26–29) e `getBankDatabase` (linhas 32–35). As subclasses `Transaction` herdam esses métodos de `Transaction` e os utilizam para ganhar acesso aos atributos `private` da classe `Transaction`.

A classe `Transaction` também declara um método `execute` `abstract` (linha 38). Não faz sentido fornecer uma implementação para esse método, pois uma transação genérica não pode ser executada. Portanto, declaramos esse método como `abstract` e forçamos cada subclasse `Transaction` a fornecer sua própria implementação concreta que executa esse tipo de transação em particular.

J.10 Classe `BalanceInquiry`

A classe `BalanceInquiry` (Figura J.9) estende a `Transaction` e representa uma transação no ATM de consulta de saldos. `BalanceInquiry` não contém nenhum atributo próprio, mas ela herda os atributos `accountNumber`, `screen` e `bankDatabase` de `Transaction`, acessíveis por meio dos métodos *get* `public` da classe `Transaction`. O construtor `BalanceInquiry` recebe os argumentos correspondentes a esses atributos e simplesmente os encaminha ao construtor da `Transaction` utilizando `super` (linha 10).

A classe `BalanceInquiry` sobrescreve o método abstrato `execute` da classe `Transaction` para fornecer uma implementação concreta (linhas 14–35) que realiza os procedimentos envolvidos em uma consulta de saldos. As linhas 17–18 obtêm as referências dos dados do banco e da tela do ATM invocando os métodos herdados da superclasse `Transaction`. As linhas 21–22 recuperam o saldo disponível da conta envolvida invocando o método `getAvailableBalance` de `bankDatabase`. Observe que a linha 22 utiliza o método herdado `getAccountNumber` para obter o número da conta do usuário atual, que ela então passa para `getAvailableBalance`. As linhas 25–26 recuperam o saldo total da conta do usuário atual. As linhas 29–34 exibem as informações sobre o saldo na tela do ATM. Lembre-se de que `displayDollarAmount` recebe um argumento `double` e gera a saída dele na tela formatado como uma quantia em dólares. Por exemplo, se `availableBalance` de um usuário for `1000.5`, a linha 31 irá gerar a saída de `$1,000.50`. Observe que a linha 34 insere uma linha em branco da saída para separar as informações do saldo da saída subseqüente (isto é, o menu principal repetido pela classe `ATM` depois de executar a `BalanceInquiry`).

J.11 Classe `Withdrawal`

A classe `Withdrawal` (Figura J.10) estende `Transaction` e representa uma transação de saque no ATM. Essa classe expande o código 'esqueleto' para essa classe desenvolvida na Figura. 10.24. Lembre-se, no diagrama de classes da Figura 10.21, de que a classe

`Withdrawal` tem um atributo, `amount`, o qual a linha 6 implementa como um campo `int`. A Figura 10.21 modela as associações entre a classe `Withdrawal` e as classes `Keypad` e `CashDispenser`, com as quais as linhas 7–8 implementam os atributos `keypad` e `cashDispenser`, respectivamente, do tipo por referência. A linha 11 declara uma constante que corresponde à opção de cancelamento no menu. Discutiremos mais adiante como a classe utiliza essa constante.

```
1  // BalanceInquiry.java
2  // Representa uma transação de consulta de saldos no ATM
3
4  public class BalanceInquiry extends Transaction
5  {
6     // Construtor de BalanceInquiry
7     public BalanceInquiry( int userAccountNumber, Screen atmScreen,
8        BankDatabase atmBankDatabase )
9     {
10       super( userAccountNumber, atmScreen, atmBankDatabase );
11    } // fim do construtor de BalanceInquiry
12
13    // realiza a transação
14    public void execute()
15    {
16       // obtém as referências ao banco de dados e tela do banco
17       BankDatabase bankDatabase = getBankDatabase();
18       Screen screen = getScreen();
19
20       // obtém o saldo disponível da conta envolvida
21       double availableBalance =
22          bankDatabase.getAvailableBalance( getAccountNumber() );
23
24       // obtém o saldo total da conta envolvida
25       double totalBalance =
26          bankDatabase.getTotalBalance( getAccountNumber() );
27
28       // exibe as informações sobre o saldo na tela
29       screen.displayMessageLine( "\nBalance Information:" );
30       screen.displayMessage( " - Available balance: " );
31       screen.displayDollarAmount( availableBalance );
32       screen.displayMessage( "\n - Total balance:     " );
33       screen.displayDollarAmount( totalBalance );
34       screen.displayMessageLine( "" );
35    } // fim do método execute
36 } // fim da classe BalanceInquiry
```

Figura J.9 A classe `BalanceInquiry` representa uma transação de consulta de saldo no ATM.

O construtor da classe `Withdrawal` (linhas 14–24) tem cinco parâmetros. Ele utiliza `super` para passar os parâmetros `userAccountNumber`, `atmScreen` e `atmBankDatabase` para o construtor da superclasse `Transaction` a fim de configurar os atributos que `Withdrawal` herda de `Transaction`. O construtor também recebe as referências `atmKeypad` e `atmCashDispenser` como parâmetros e as especifica para os atributos `keypad` e `cashDispenser` do tipo por referência.

A classe `Withdrawal` sobrescreve o método `execute` abstrato de `Transaction` com uma implementação concreta (linhas 27–84) que realiza os procedimentos envolvidos em um saque. A linha 29 declara e inicializa uma variável `boolean cashDispensed` local. Essa variável indica se cédulas foram entregues (se a transação foi completada com sucesso) e inicialmente é `false`. A linha 30 declara a variável `double availableBalance` local, que armazenará o saldo disponível do usuário durante uma transação de saque. As linhas 33–34 obtêm as referências dos dados do banco e da tela do ATM invocando os métodos herdados da superclasse `Transaction`.

As linhas 37–82 contêm uma instrução `do...while` que executa seu corpo até que cédulas sejam entregues (até `cashDispensed` tornar-se `true`) ou até o usuário escolher cancelar (nesse caso, o loop termina). Utilizamos esse loop continuamente para retornar o usuário ao início da transação se ocorrer um erro (isto é, a quantia de saque solicitada é maior do que o saldo disponível do usuário ou maior do que a quantia de cédulas no dispensador de cédulas). A linha 40 exibe um menu das quantias de saques e obtém uma seleção de

usuário chamando método utilitário `private` `displayMenuOfAmounts` (declarado nas linhas 88–132). Esse método exibe o menu das quantias e retorna uma quantia de saque `int` ou uma constante `int` `CANCELED` para indicar que o usuário optou por cancelar a transação.

```
1  // Withdrawal.java
2  // Representa uma transação de saque no ATM
3
4  public class Withdrawal extends Transaction
5  {
6     private int amount; // quantia a sacar
7     private Keypad keypad; // referência ao teclado numérico
8     private CashDispenser cashDispenser; // referência ao dispensador de cédulas
9
10    // constante que corresponde à opção cancelar no menu
11    private final static int CANCELED = 6;
12
13    // Construtor de Withdrawal
14    public Withdrawal( int userAccountNumber, Screen atmScreen,
15       BankDatabase atmBankDatabase, Keypad atmKeypad,
16       CashDispenser atmCashDispenser )
17    {
18       // inicializa as variáveis da superclasse
19       super( userAccountNumber, atmScreen, atmBankDatabase );
20
21       // inicializa as referências ao teclado numérico e ao dispensador de cédulas
22       keypad = atmKeypad;
23       cashDispenser = atmCashDispenser;
24    } // fim do construtor de Withdrawal
25
26    // realiza a transação
27    public void execute()
28    {
29       boolean cashDispensed = false; // cédulas ainda não foram entregues
30       double availableBalance; // quantia disponível para saque
31
32       // obtém as referências ao banco de dados e tela do banco
33       BankDatabase bankDatabase = getBankDatabase();
34       Screen screen = getScreen();
35
36       // faz um loop até as cédulas serem entregues ou o usuário cancelar
37       do
38       {
39          // obtém a quantia de um saque escolhida pelo usuário
40          amount = displayMenuOfAmounts();
41
42          // verifica se o usuário escolheu uma quantia de saque ou cancelou
43          if ( amount != CANCELED )
44          {
45             // obtém o saldo disponível na conta envolvida
46             availableBalance =
47                bankDatabase.getAvailableBalance( getAccountNumber() );
48
49             // verifica se o usuário tem dinheiro suficiente na conta
50             if ( amount <= availableBalance )
51             {
52                // verifica se o dispensador de cédulas tem cédulas suficientes
```

Figura J.10 A classe `Withdrawal` representa uma transação de saque no ATM. (Parte 1 de 3.)

```
53                if ( cashDispenser.isSufficientCashAvailable( amount ) )
54                {
55                   // atualiza a conta envolvida para refletir a retirada/saque
56                   bankDatabase.debit( getAccountNumber(), amount );
57
58                   cashDispenser.dispenseCash( amount ); // entrega cédulas
59                   cashDispensed = true; // cédulas foram entregues
60
61                   // instrui o usuário a pegar as cédulas
62                   screen.displayMessageLine( "\nYour cash has been" +
63                      " dispensed. Please take your cash now." );
64                } // fim do if
65                else // o dispensador de cédulas não tem cédulas suficientes
66                   screen.displayMessageLine(
67                      "\nInsufficient cash available in the ATM." +
68                      "\n\nPlease choose a smaller amount." );
69             } // fim do if
70             else // não há dinheiro suficiente disponível na conta do usuário
71             {
72                screen.displayMessageLine(
73                   "\nInsufficient funds in your account." +
74                   "\n\nPlease choose a smaller amount." );
75             } // fim de else
76          } // fim do if
77          else // o usuário escolheu a opção cancelar no menu
78          {
79             screen.displayMessageLine( "\nCanceling transaction..." );
80             return; // retorna ao menu principal porque o usuário cancelou
81          } // fim de else
82       } while ( !cashDispensed );
83
84    } // fim do método execute
85
86    // exibe um menu de quantias de saques e a opção para cancelar;
87    // retorna a quantia escolhida ou 0 se o usuário escolher cancelar
88    private int displayMenuOfAmounts()
89    {
90       int userChoice = 0; // variável local para armazenar o valor de retorno
91
92       Screen screen = getScreen(); // obtém referência de tela
93
94       // array de quantias que correspondem aos números no menu
95       int amounts[] = { 0, 20, 40, 60, 100, 200 };
96
97       // faz um loop enquanto nenhuma escolha válida for feita
98       while ( userChoice == 0 )
99       {
100         // exibe o menu
101         screen.displayMessageLine( "\nWithdrawal Menu:" );
102         screen.displayMessageLine( "1 - $20" );
103         screen.displayMessageLine( "2 - $40" );
104         screen.displayMessageLine( "3 - $60" );
105         screen.displayMessageLine( "4 - $100" );
106         screen.displayMessageLine( "5 - $200" );
107         screen.displayMessageLine( "6 - Cancel transaction" );
```

Figura J.10 A classe `Withdrawal` representa uma transação de saque no ATM. (Parte 2 de 3.)

```
108          screen.displayMessage( "\nChoose a withdrawal amount: " );
109
110          int input = keypad.getInput(); // obtém a entrada do usuário pelo teclado
111
112          // determina como prosseguir com base no valor de entrada
113          switch ( input )
114          {
115             case 1: // se o usuário escolheu uma quantia de saque
116             case 2: // (isto é, escolheu a opção 1, 2, 3, 4 ou 5), retorna a
117             case 3: // quantia correspondente do array de quantias
118             case 4:
119             case 5:
120                userChoice = amounts[ input ]; // salva a escolha do usuário
121                break;
122             case CANCELED: // o usuário escolheu cancelar
123                userChoice = CANCELED; // salva a escolha do usuário
124                break;
125             default: // o usuário não inseriu um valor ente 1 e 6
126                screen.displayMessageLine(
127                   "\nInvalid selection. Try again." );
128          } // fim de switch
129       } // fim do while
130
131       return userChoice; // retorna a quantia de saque ou CANCELADA
132    } // fim do método displayMenuOfAmounts
133 } // fim da classe Withdrawal
```

Figura J.10 A classe `Withdrawal` representa uma transação de saque no ATM. (Parte 3 de 3.)

O método `displayMenuOfAmounts` (linhas 88–132) primeiro declara a variável local `userChoice` (inicialmente `0`) para armazenar o valor que o método retornará (linha 90). A linha 92 obtém uma referência à tela chamando o método `getScreen` herdado da superclasse `Transaction`. A linha 95 declara um array de inteiros das quantias de saque que correspondem às quantias exibidas no menu de saque. Ignoramos o primeiro elemento no array (índice `0`) porque o menu não tem nenhuma opção 0. A instrução `while` nas linhas 98–129 é repetida até `userChoice` assumir um valor além de `0`. Veremos mais adiante que isso ocorre quando o usuário faz uma seleção válida no menu. As linhas 101–108 exibem o menu de saque na tela e solicitam ao usuário inserir uma escolha. A linha 110 obtém o inteiro de `input` pelo teclado. A instrução `switch` nas linhas 113–128 determina como prosseguir com base na entrada do usuário. Se o usuário selecionar um número entre `1` e `5`, a linha 120 configurará `userChoice` como o valor do elemento em `amounts` no índice `input`. Por exemplo, se o usuário inserir `3` para sacar US$ 60, a linha 120 configurará `userChoice` como o valor de `amounts[ 3 ]` (isto é, `60`). A linha 120 termina a instrução `switch`. A variável `userChoice` não mais é igual a `0`, assim o `while` nas linhas 98–129 termina e a linha 131 retorna `userChoice`. Se o usuário selecionar a opção cancelar no menu, as linhas 123–124 executarão, configurando `userChoice` como `CANCELED` e fazendo com que o método retorne esse valor. Se o usuário não inserir uma seleção válida de menu, as linhas 126–127 exibirão uma mensagem de erro e o usuário será retornado ao menu de saque.

A instrução `if` na linha 43 no método `execute` determina se o usuário selecionou uma quantia de saque ou escolheu cancelar. Se o usuário cancelar, as linhas 79–80 executarão e exibirão uma mensagem apropriada para o usuário antes de retornar o controle ao método que chama (o método `performTransactions` da classe `ATM`). Se o usuário escolheu uma quantia de saque, as linhas 46–47 recuperam o saldo disponível em `Account` do usuário atual e o armazenam na variável `availableBalance`. Em seguida, a instrução `if` na linha 50 determina se a quantia selecionada é menor ou igual ao saldo disponível do usuário. Se não for, as linhas 72–74 exibem uma mensagem de erro apropriada. O controle então continua até o final da instrução `do...while` e os loops repetem porque `cashDispensed` ainda é `false`. Se o saldo do usuário for suficientemente alto, a instrução `if` na linha 53 determinará se o dispensador de cédulas tem cédulas suficientes para satisfazer a solicitação de saque invocando o método `isSufficientCashAvailable` de `cashDispenser`. Se esse método retornar `false`, as linhas 66–68 exibirão uma mensagem de erro apropriada e a instrução `do...while` será repetida. Se houver cédulas suficientes disponíveis, então os requisitos para o saque serão atendidos e a linha 56 debitará `amount` da conta do usuário no banco de dados. As linhas 58–59 então instruem o dispensador de cédulas a entregar as cédulas ao usuário e configuram `cashDispensed` como `true`. Por fim, as linhas 62–63 exibem uma mensagem para o usuário de que as cédulas foram entregues. Como `cashDispensed` agora é `true`, o controle continua depois de `do...while`. Nenhuma instrução adicional aparece abaixo do loop, portanto o método retorna o controle à classe `ATM`.

J.12 Classe Deposit

A classe `Deposit` (Figura J.11) estende `Transaction` e representa uma transação de depósito no ATM. Lembre-se, no diagrama de classes da Figura 10.22, de que a classe `Deposit` tem um atributo `amount`, que a linha 6 implementa como um campo `int`. As linhas 7–8 criam os atributos `keypad` e `depositSlot` do tipo por referência que implementam as associações entre a classe `Deposit` e as classes `Keypad` e `DepositSlot` modeladas na Figura 10.21. A linha 9 declara uma constante `CANCELED` que corresponde ao valor que um usuário insere para cancelar. Discutiremos mais adiante como a classe utiliza essa constante.

Como ocorre com a classe `Withdrawal`, a classe `Deposit` contém um construtor (linhas 12–22) que passa três parâmetros para o construtor da superclasse `Transaction` utilizando `super`. O construtor também contém os parâmetros `atmKeypad` e `atmDepositSlot`, que ele especifica para os atributos correspondentes (linhas 20–21).

O método `execute` (linhas 25–65) sobrescreve o método abstrato `execute` na superclasse `Transaction` por uma implementação concreta que realiza os procedimentos necessários em uma transação de depósito. As linhas 27–28 obtêm as referências ao banco de dados e à tela. A linha 30 solicita que o usuário insira uma quantia de depósito invocando o método utilitário `private promptForDepositAmount` (declarado nas linhas 68–84) e configura o atributo `amount` como o valor retornado. O método `promptForDepositAmount` solicita que o usuário insira uma quantia de depósito como um número inteiro dos centavos (visto que o teclado numérico do ATM não contém um ponto de fração decimal; isso é compatível com muitos ATMS reais) e retorna o valor de `double` que representa a quantia em dólares a ser depositada.

```
1  // Deposit.java
2  // Representa uma transação de depósito no ATM
3
4  public class Deposit extends Transaction
5  {
6     private double amount; // quantia a depositar
7     private Keypad keypad; // referência ao teclado numérico
8     private DepositSlot depositSlot; // referência à abertura para depósito
9     private final static int CANCELED = 0; // constante para a opção de cancelamento
10
11    // Construtor de Deposit
12    public Deposit( int userAccountNumber, Screen atmScreen,
13       BankDatabase atmBankDatabase, Keypad atmKeypad,
14       DepositSlot atmDepositSlot )
15    {
16       // inicializa as variáveis da superclasse
17       super( userAccountNumber, atmScreen, atmBankDatabase );
18
19       // inicializa as referências a teclado e abertura para depósito
20       keypad = atmKeypad;
21       depositSlot = atmDepositSlot;
22    } // fim do construtor de Deposit
23
24    // realiza a transação
25    public void execute()
26    {
27       BankDatabase bankDatabase = getBankDatabase(); // obtém a referência
28       Screen screen = getScreen(); // obtém a referência
29
30       amount = promptForDepositAmount(); // obtém a quantia de depósito do usuário
31
32       // verifica se usuário inseriu uma quantia de depósito ou cancelou
33       if ( amount != CANCELED )
34       {
35          // solicita o envelope de depósito contendo a quantia especificada
36          screen.displayMessage(
37             "\nPlease insert a deposit envelope containing " );
38          screen.displayDollarAmount( amount );
```

Figura J.11 A classe `Deposit` representa uma transação de depósito no ATM. (Parte 1 de 2.)

```
39          screen.displayMessageLine( "." );
40
41          // recebe o envelope de depósito
42          boolean envelopeReceived = depositSlot.isEnvelopeReceived();
43
44          // verifica se envelope de depósito foi recebido
45          if ( envelopeReceived )
46          {
47             screen.displayMessageLine( "\nYour envelope has been " +
48                "received.\nNOTE: The money just deposited will not " +
49                "be available until we verify the amount of any " +
50                "enclosed cash and your checks clear." );
51
52             // credita na conta para refletir o depósito
53             bankDatabase.credit( getAccountNumber(), amount );
54          } // fim do if
55          else // envelope de depósito não foi recebido
56          {
57             screen.displayMessageLine( "\nYou did not insert an " +
58                "envelope, so the ATM has canceled your transaction." );
59          } // fim de else
60       } // fim do if
61       else // o usuário cancelou em vez de inserir uma quantia
62       {
63          screen.displayMessageLine( "\nCanceling transaction..." );
64       } // fim de else
65    } // fim do método execute
66
67    // solicita que o usuário insira uma quantia de depósito em centavos
68    private double promptForDepositAmount()
69    {
70       Screen screen = getScreen(); // obtém a referência à tela
71
72       // exibe a solicitação
73       screen.displayMessage( "\nPlease enter a deposit amount in " +
74          "CENTS (or 0 to cancel): " );
75       int input = keypad.getInput(); // recebe a entrada da quantia do depósito
76
77       // verifica se o usuário cancelou ou inseriu uma quantia válida
78       if ( input == CANCELED )
79          return CANCELED;
80       else
81       {
82          return ( double ) input / 100; // retorna a quantia em dólares
83       } // fim de else
84    } // fim do método promptForDepositAmount
85 } // fim da classe Deposit
```

Figura J.11 A classe `Deposit` representa uma transação de depósito no ATM. (Parte 2 de 2.)

A linha 70 no método `promptForDepositAmount` obtém uma referência à tela do ATM. As linhas 73–74 exibem uma mensagem na tela que solicita ao usuário inserir uma quantia de depósito como um número de centavos ou '0' para cancelar a transação. A linha 75 recebe a entrada do usuário do teclado. A instrução `if` nas linhas 78–83 determina se o usuário inseriu uma quantia de depósito real ou optou por cancelar. Se o usuário escolher cancelar, a linha 79 retornará a constante `CANCELED`. Caso contrário, a linha 82 retornará a quantia de depósito depois de converter do número de centavos para uma quantia em dólares fazendo uma coerção de `input` para um `double` e então dividindo por `100`. Por exemplo, se o usuário inserir `125` como o número de centavos, a linha 82 retornará `125.0` dividido por `100` ou `1.25` — 125 centavos é US$ 1,25.

A instrução `if` nas linhas 33–64 no método `execute` determina se o usuário escolheu cancelar a transação em vez de inserir uma quantia de depósito. Se o usuário cancelar, a linha 63 exibirá uma mensagem apropriada e o método retornará. Se o usuário inserir uma quantia de depósito, as linhas 36–39 irão instruir o usuário a inserir um envelope de depósito com a quantia correta. Lembre-se de que o método `displayDollarAmount` de `Screen` gera a saída de um `double` formatado como uma quantia em dólares.

A linha 42 configura uma variável `boolean` local como o valor retornado pelo método `isEnvelopeReceived` de `depositSlot`, indicando se um envelope de depósito foi recebido. Lembre-se de que codificamos o método `isEnvelopeReceived` (linhas 8–11 da Figura J.5) para sempre retornar `true`, pois estamos simulando a funcionalidade da abertura para depósito e supomos que o usuário sempre insere um envelope. Entretanto, codificamos o método `execute` da classe `Deposit` para testar a possibilidade de o usuário não inserir um envelope — boa engenharia de software demanda que os programas são responsáveis por todos os possíveis valores de retorno. Portanto, a classe `Deposit` está preparada para futuras versões do `isEnvelopeReceived` que poderiam retornar `false`. As linhas 47–53 executam se a abertura de depósito receber um envelope. As linhas 47–50 exibem uma mensagem apropriada para o usuário. A linha 53 então credita a quantia de depósito na conta do usuário no banco de dados. As linhas 57–58 executarão se a abertura de depósito não receber um envelope de depósito. Nesse caso, exibimos uma mensagem ao usuário indicando que o ATM cancelou a transação. O método então retorna sem modificar a conta do usuário.

J.13 Classe ATMCaseStudy

A classe `ATMCaseStudy` (Figura J.12) é uma classe simples que permite iniciar ou 'ativar' o ATM e testar a implementação do nosso modelo do sistema ATM. O método `main` da classe `ATMCaseStudy` (linhas 7–11) não faz nada além de instanciar um novo objeto `ATM` chamado `theATM` (linha 9) e invocar seu método `run` (linha 10) para iniciar o ATM.

J.14 Conclusão

Parabéns por ter completado todo o estudo de caso de engenharia de software do ATM! Esperamos que essa experiência tenha sido valiosa e que tenha reforçado muitos dos conceitos que você aprendeu nos capítulos 1 a 10. Gostaríamos, sinceramente, de ouvir seus comentários, críticas e sugestões. Você pode nos contatar em `deitel@deitel.com`. Responderemos prontamente.

```
1  // ATMCaseStudy.java
2  // Programa de driver para o estudo de caso do ATM
3
4  public class ATMCaseStudy
5  {
6     // método main cria e executa o ATM
7     public static void main( String[] args )
8     {
9        ATM theATM = new ATM();
10       theATM.run();
11    } // fim de main
12 } // fim da classe ATMCaseStudy
```

Figura J.12 A classe `ATMCaseStudy.java` inicia o ATM.