WORKED EXAMPLE 12.1 Simulating an Automatic Teller Machine

In this Worked Example, we apply the object-oriented design methodology to the simulation of an automatic teller machine (ATM).

Problem Statement Simulate an ATM that handles checking and savings accounts. Provide both a console-based and graphical user interface.

© Mark Evans/iStockphoto.

Step 1 Gather requirements.

The purpose of this project is to simulate an automatic teller machine. The ATM is used by the customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer also has a customer number and a personal identification number (PIN); both are required to gain access to the accounts. (In a real ATM, the customer number would be recorded on the magnetic strip of the ATM card. In this simulation, the customer will need to type it in.) With the ATM, customers can select an account (checking or savings). The balance of the selected account is displayed. Then the customer can deposit and withdraw money. This process is repeated until the customer chooses to exit.

The details of the user interaction depend on the user interface that we choose for the simulation. We will develop two separate interfaces: a graphical interface that closely mimics an actual ATM (see Figure 9), and a text-based interface that allows you to test the ATM and bank classes without being distracted by GUI programming.

In the GUI interface, the ATM has a keypad to enter numbers, a display to show messages, and a set of buttons, labeled A, B, and C, whose function depends on the state of the machine.

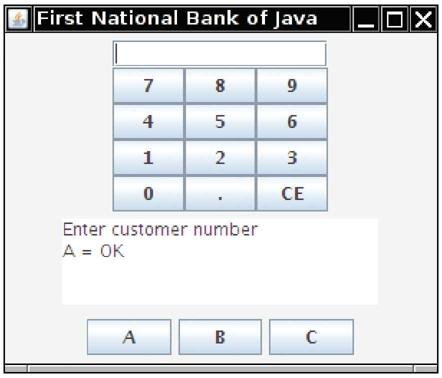
Specifically, the user interaction is as follows. When the ATM starts up, it expects a user to enter a customer number. The display shows the following message:

```
Enter customer number
A = OK
```

The user enters the customer number on the keypad and presses the A button. The display message changes to

```
Enter PIN
A = OK
```

Next, the user enters the PIN and presses the A button again. If the customer number and ID match those of one of the customers in the bank, then the customer can proceed. If not, the user is again prompted to enter the customer number.



```
Enter customer number
A = OK
```

Figure 9
Graphical User Interface for the Automatic Teller Machine

If the customer has been authorized to use the system, then the display message changes to

```
Select Account
A = Checking
B = Savings
C = Exit
```

If the user presses the C button, the ATM reverts to its original state and asks the next user to enter a customer number.

If the user presses the A or B buttons, the ATM remembers the selected account, and the display message changes to

```
Balance = balance of selected account
Enter amount and select transaction
A = Withdraw
B = Deposit
C = Cancel
```

If the user presses the A or B buttons, the value entered in the keypad is withdrawn from or deposited into the selected account. (This is just a simulation, so no money is dispensed and no deposit is accepted.) Afterward, the ATM reverts to the preceding state, allowing the user to select another account or to exit.

If the user presses the C button, the ATM reverts to the preceding state without executing any transaction.

In the text-based interaction, we read input from `System.in` instead of the buttons. Here is a typical dialog:

```
Enter customer number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

In our solution, only the user interface classes are affected by the choice of user interface. The remainder of the classes can be used for both solutions—they are decoupled from the user interface.

Because this is a simulation, the ATM does not actually communicate with a bank. It simply loads a set of customer numbers and PINs from a file. All accounts are initialized with a zero balance.

Step 2 Use CRC cards to find classes, responsibilities, and collaborators.

We will again follow the recipe of Section 12.2 and show how to discover classes, responsibilities, and relationships and how to obtain a detailed design for the ATM program.

Recall that the first rule for finding classes is "Look for nouns in the problem description". Here is a list of the nouns:

```
ATM
User
Keypad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank
```

Of course, not all of these nouns will become names of classes, and we may yet discover the need for classes that aren't in this list, but it is a good start.

Users and customers represent the same concept in this program. Let's use a class `Customer`. A customer has two bank accounts, and we will require that a `Customer` object should be able to locate these accounts. (Another possible design would make the `Bank` class responsible for locating the accounts of a given customer—see Exercise E12.6.)

A customer also has a customer number and a PIN. We can, of course, require that a customer object give us the customer number and the PIN. But perhaps that isn't so secure. Instead, simply require that a customer object, when given a customer number and a PIN, will tell us whether it matches its own information or not.

Customer	
get accounts	
match number and PIN	

A bank contains a collection of customers. When a user walks up to the ATM and enters a customer number and PIN, it is the job of the bank to find the matching customer. How can the bank do this? It needs to check for each customer whether its customer number and PIN match. Thus, it needs to call the *match number and PIN* method of the `Customer` class that we just discovered. Because the *find customer* method calls a `Customer` method, it collaborates with the `Customer` class. We record that fact in the right-hand column of the CRC card.

When the simulation starts up, the bank must also be able to read customer information from a file.

Bank	
find customer	`Customer`
read customers	

The `BankAccount` class is our familiar class with methods to get the balance and to deposit and withdraw money.

In this program, there is nothing that distinguishes checking accounts from savings accounts. The ATM does not add interest or deduct fees. Therefore, we decide not to implement separate subclasses for checking and savings accounts.

Finally, we are left with the ATM class itself. An important notion of the ATM is the **state**. The current machine state determines the text of the prompts and the function of the buttons. For example, when you first log in, you use the A and B buttons to select an account. Next, you use the same buttons to choose between deposit and withdrawal. The ATM must remember the current state so that it can correctly interpret the buttons.

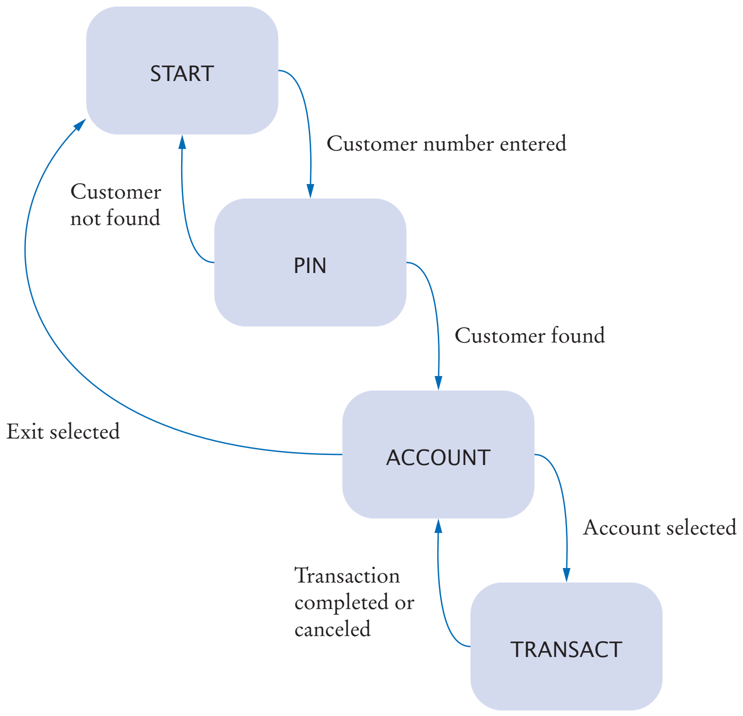


Figure 10 State Diagram for the ATM Class

There are four states:

1. START: Enter customer ID
2. PIN: Enter PIN
3. ACCOUNT: Select account
4. TRANSACT: Select transaction

To understand how to move from one state to the next, it is useful to draw a **state diagram** (Figure 10). The UML notation has standardized shapes for state diagrams. Draw states as rectangles with rounded corners. Draw state changes as arrows, with labels that indicate the reason for the change.

The user must type a valid customer number and PIN. Then the ATM can ask the bank to find the customer. This calls for a *select customer* method. It collaborates with the bank, asking the bank for the customer that matches the customer number and PIN. Next, there must be a *select account* method that asks the current customer for the checking or savings account. Finally, the ATM must carry out the selected transaction on the current account.

ATM	
manage state	`Customer`
select customer	`Bank`
select account	`BankAccount`
execute transaction	

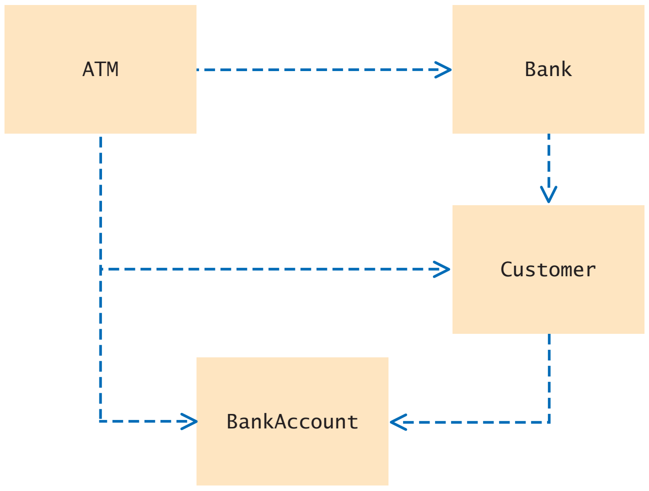
Of course, discovering these classes and methods was not as neat and orderly as it appears from this discussion. When I designed these classes for this book, it took me several trials and many torn cards to come up with a satisfactory design. It is also important to remember that there is seldom one best design.

This design has several advantages. The classes describe clear concepts. The methods are sufficient to implement all necessary tasks. (I mentally walked through every ATM usage scenario to verify that.) There are not too many collaboration dependencies between the classes. Thus, I was satisfied with this design and proceeded to the next step.

Step 3 Use UML diagrams to record class relationships.

To draw the dependencies, use the "collaborator" columns from the CRC cards. Looking at those columns, you find that the dependencies are as follows:

- `ATM` knows about `Bank`, `Customer`, and `BankAccount`.
- `Bank` knows about `Customer`.
- `Customer` knows about `BankAccount`.



It is easy to see some of the aggregation relationships. A bank has customers, and each customer has two bank accounts.

Does the `ATM` class aggregate `Bank`? To answer this question, ask yourself whether an ATM object needs to store a reference to a bank object. Does it need to locate the same bank object across multiple method calls? Indeed it does. Therefore, aggregation is the appropriate relationship.

Does an ATM aggregate customers? Clearly, the ATM is not responsible for storing all of the bank's customers. That's the bank's job. But in our design, the ATM remembers the *current* customer. If a customer has logged in, subsequent commands refer to the same customer. The ATM needs to either store a reference to the customer, or ask the bank to look up the object whenever it needs the current customer. It is a design decision: either store the object, or look it up when needed. We will decide to store the current customer object. That is, we will use aggregation. Note that the choice of aggregation is not an automatic consequence of the problem description—it is a design decision.

Similarly, we will decide to store the current bank account (checking or savings) that the user selects. Therefore, we have an aggregation relationship between `ATM` and `BankAccount`.

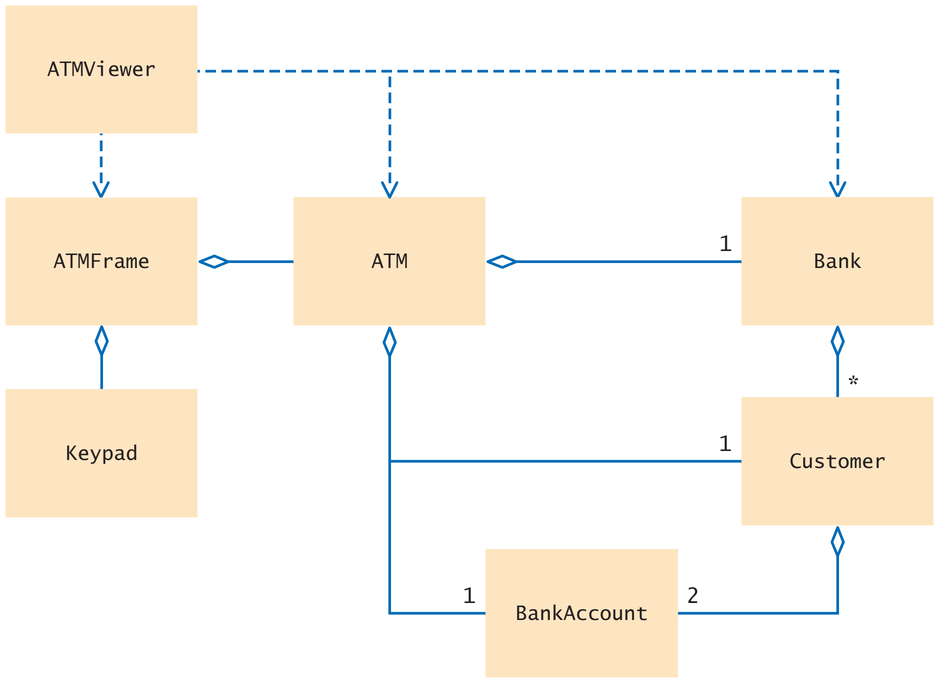


Figure 11 Relationships Between the ATM Classes

Figure 11 shows the relationships between these classes, using the graphical user interface. (The console user interface uses a single class `ATMSimulator` instead of the `ATMFrame`, `ATMViewer`, and `Keypad` classes.)

The class diagram is a good tool to visualize dependencies. Look at the GUI classes. They are completely independent from the rest of the ATM system. You can replace the GUI with a console interface, and you can take out the `Keypad` class and use it in another application. Also, the `Bank`, `BankAccount`, and `Customer` classes, although dependent on each other, don't know anything about the `ATM` class. That makes sense—you can have banks without ATMs. As you can see, when you analyze relationships, you look for both the absence and presence of relationships.

Step 4 Use `javadoc` to document method behavior.

Now you are ready for the final step of the design phase: documenting the classes and methods that you discovered. Here is a part of the documentation for the `ATM` class:

```
/**
   An ATM that accesses a bank.
*/
public class ATM
{
   . . .
   /**
      Constructs an ATM for a given bank.
      @param aBank the bank to which this ATM connects
   */
   public ATM(Bank aBank) { }

   /**
      Sets the current customer number
      and sets state to PIN.
      (Precondition: state is START)
      @param number the customer number
   */
   public void setCustomerNumber(int number) { }

   /**
      Finds customer in bank.
      If found sets state to ACCOUNT, else to START.
      (Precondition: state is PIN)
      @param pin the PIN of the current customer
   */
   public void selectCustomer(int pin) { }

   /**
      Sets current account to checking or savings. Sets
      state to TRANSACT.
      (Precondition: state is ACCOUNT or TRANSACT)
      @param account one of CHECKING or SAVINGS
   */
   public void selectAccount(int account) { }

   /**
      Withdraws amount from current account.
      (Precondition: state is TRANSACT)
      @param value the amount to withdraw
   */
   public void withdraw(double value) { }
   . . .
}
```

Then run the javadoc utility to turn this documentation into HTML format.

For conciseness, we omit the documentation of the other classes, but they are shown at the end of this example.

Step 5 Implement your program.

Finally, the time has come to implement the ATM simulator. The implementation phase is very straightforward and should take *much less time than the design phase.*

A good strategy for implementing the classes is to go "bottom-up". Start with the classes that don't depend on others, such as Keypad and BankAccount. Then implement a class such as Customer that depends only on the BankAccount class. This "bottom-up" approach allows you to test your classes individually. You will find the implementations of these classes at the end of this section.

The most complex class is the ATM class. In order to implement the methods, you need to declare the necessary instance variables. From the class diagram, you can tell that the ATM has a bank object. It becomes an instance variable of the class:

```
public class ATM
{
   private Bank theBank;
   . . .
}
```

From the description of the ATM states, it is clear that we require additional instance variables to store the current state, customer, and bank account:

```
public class ATM
{
   private int state;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   . . .
}
```

Most methods are very straightforward to implement. Consider the `selectCustomer` method. From the design documentation, we have the description

```
/**
   Finds customer in bank.
   If found sets state to ACCOUNT, else to START.
   (Precondition: state is PIN)
   @param pin the PIN of the current customer
*/
```

This description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)
{
   currentCustomer = theBank.findCustomer(customerNumber, pin);
   if (currentCustomer == null)
   {
      state = START;
   }
   else
   {
      state = ACCOUNT;
   }
}
```

We won't go through a method-by-method description of the ATM program. You should take some time and compare the actual implementation to the CRC cards and the UML diagram.

worked_example_1/BankAccount.java

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private double balance;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /**
      Deposits money into the account.
      @param amount the amount of money to withdraw
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /**
      Withdraws money from the account.
      @param amount the amount of money to deposit
   */
   public void withdraw(double amount)
   {
      balance = balance - amount;
   }

   /**
      Gets the account balance.
      @return the account balance
   */
   public double getBalance()
   {
      return balance;
   }
}
```

worked_example_1/Customer.java

```
/**
   A bank customer with a checking and a savings account.
*/
public class Customer
{
   private int customerNumber;
   private int pin;
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;

   /**
      Constructs a customer with a given number and PIN.
      @param aNumber the customer number
      @param aPin the personal identification number
   */
   public Customer(int aNumber, int aPin)
   {
      customerNumber = aNumber;
      pin = aPin;
      checkingAccount = new BankAccount();
      savingsAccount = new BankAccount();
   }

   /**
      Tests if this customer matches a customer number
      and PIN.
      @param aNumber a customer number
      @param aPin a personal identification number
      @return true if the customer number and PIN match
   */
   public boolean match(int aNumber, int aPin)
   {
      return customerNumber == aNumber && pin == aPin;
   }

   /**
```

```java
      Gets the checking account of this customer.
      @return the checking account
   */
   public BankAccount getCheckingAccount()
   {
      return checkingAccount;
   }

   /**
      Gets the savings account of this customer.
      @return the checking account
   */
   public BankAccount getSavingsAccount()
   {
      return savingsAccount;
   }
}
```

worked_example_1/Bank.java

```java
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
   A bank contains customers.
*/
public class Bank
{
   private ArrayList<Customer> customers;

   /**
      Constructs a bank with no customers.
   */
   public Bank()
   {
      customers = new ArrayList<Customer>();
   }

   /**
      Reads the customer numbers and pins.
      @param filename the name of the customer file
   */
   public void readCustomers(String filename)
         throws IOException
   {
      Scanner in = new Scanner(new File(filename));
      while (in.hasNext())
      {
         int number = in.nextInt();
         int pin = in.nextInt();
         Customer c = new Customer(number, pin);
         addCustomer(c);
      }
      in.close();
   }

   /**
```

```
      Adds a customer to the bank.
      @param c the customer to add
   */
   public void addCustomer(Customer c)
   {
      customers.add(c);
   }

   /**
      Finds a customer in the bank.
      @param aNumber a customer number
      @param aPin a personal identification number
      @return the matching customer, or null if no customer
      matches
   */
   public Customer findCustomer(int aNumber, int aPin)
   {
      for (Customer c : customers)
      {
         if (c.match(aNumber, aPin))
         {
            return c;
         }
      }
      return null;
   }
}
```

worked_example_1/ATM.java

```
/**
   An ATM that accesses a bank.
*/
public class ATM
{
   public static final int CHECKING = 1;
   public static final int SAVINGS = 2;

   private int state;
   private int customerNumber;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;

   public static final int START = 1;
   public static final int PIN = 2;
   public static final int ACCOUNT = 3;
   public static final int TRANSACT = 4;

   /**
      Constructs an ATM for a given bank.
      @param aBank the bank to which this ATM connects
   */
   public ATM(Bank aBank)
   {
      theBank = aBank;
      reset();
   }

   /**
      Resets the ATM to the initial state.
   */
   public void reset()
   {
      customerNumber = -1;
      currentAccount = null;
      state = START;
   }

   /**
      Sets the current customer number
      and sets state to PIN.
      (Precondition: state is START)
      @param number the customer number
   */
   public void setCustomerNumber(int number)
   {
      customerNumber = number;
      state = PIN;
   }

   /**
      Finds customer in bank.
      If found sets state to ACCOUNT, else to START.
      (Precondition: state is PIN)
      @param pin the PIN of the current customer
   */
   public void selectCustomer(int pin)
   {
      currentCustomer
         = theBank.findCustomer(customerNumber, pin);
      if (currentCustomer == null)
      {
         state = START;
      }
      else
      {
         state = ACCOUNT;
      }
   }

   /**
      Sets current account to checking or savings. Sets
      state to TRANSACT.
      (Precondition: state is ACCOUNT or TRANSACT)
      @param account one of CHECKING or SAVINGS
   */
   public void selectAccount(int account)
   {
      if (account == CHECKING)
      {
         currentAccount = currentCustomer.getCheckingAccount();
      }
      else
      {
         currentAccount = currentCustomer.getSavingsAccount();
      }
      state = TRANSACT;
   }

   /**
      Withdraws amount from current account.
      (Precondition: state is TRANSACT)
      @param value the amount to withdraw
   */
   public void withdraw(double value)
   {
      currentAccount.withdraw(value);
   }

   /**
      Deposits amount to current account.
      (Precondition: state is TRANSACT)
      @param value the amount to deposit
   */
   public void deposit(double value)
   {
      currentAccount.deposit(value);
   }

   /**
      Gets the balance of the current account.
      (Precondition: state is TRANSACT)
      @return the balance
   */
   public double getBalance()
   {
      return currentAccount.getBalance();
   }

   /**
      Moves back to the previous state.
   */
   public void back()
   {
      if (state == TRANSACT)
      {
         state = ACCOUNT;
      }
      else if (state == ACCOUNT)
      {
         state = PIN;
      }
      else if (state == PIN)
      {
         state = START;
      }
   }

   /**
      Gets the current state of this ATM.
      @return the current state
   */
   public int getState()
   {
      return state;
   }
}
```

The following class implements a console-based user interface for the ATM.

worked_example_1/ATMSimulator.java

```java
import java.io.IOException;
import java.util.Scanner;

/**
   A text-based simulation of an automatic teller machine.
*/
public class ATMSimulator
{
   public static void main(String[] args)
   {
      ATM theATM;
      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch (IOException e)
      {
         System.out.println("Error opening accounts file.");
         return;
      }

      Scanner in = new Scanner(System.in);

      while (true)
      {
         int state = theATM.getState();
         if (state == ATM.START)
         {
            System.out.print("Enter customer number: ");
            int number = in.nextInt();
            theATM.setCustomerNumber(number);
         }
         else if (state == ATM.PIN)
         {
            System.out.print("Enter PIN: ");
            int pin = in.nextInt();
            theATM.selectCustomer(pin);
         }
         else if (state == ATM.ACCOUNT)
         {
            System.out.print("A=Checking, B=Savings, C=Quit: ");
            String command = in.next();
            if (command.equalsIgnoreCase("A"))
            {
               theATM.selectAccount(ATM.CHECKING);
            }
            else if (command.equalsIgnoreCase("B"))
            {
               theATM.selectAccount(ATM.SAVINGS);
            }
            else if (command.equalsIgnoreCase("C"))
            {
               theATM.reset();
            }
            else
            {
               System.out.println("Illegal input!");
            }
         }
         else if (state == ATM.TRANSACT)
         {
            System.out.println("Balance=" + theATM.getBalance());
            System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
            String command = in.next();
            if (command.equalsIgnoreCase("A"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.deposit(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("B"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.withdraw(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("C"))
            {
               theATM.back();
            }
            else
            {
               System.out.println("Illegal input!");
            }
         }
      }
   }
}
```

Program Run

```
Enter customer number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
. . .
```

Here are the user-interface classes for the GUI version of the user interface.

worked_example_1/KeyPad.java

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;
```

```

/**
   A component that lets the user enter a number, using
   a button pad labeled with digits.
*/
public class KeyPad extends JPanel
{
   private JPanel buttonPanel;
   private JButton clearButton;
   private JTextField display;

   /**
      Constructs the keypad panel.
   */
   public KeyPad()
   {
      setLayout(new BorderLayout());

      // Add display field

      display = new JTextField();
      add(display, "North");

      // Make button panel

      buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(4, 3));

      // Add digit buttons

      addButton("7");
      addButton("8");
      addButton("9");
      addButton("4");
      addButton("5");
      addButton("6");
      addButton("1");
      addButton("2");
      addButton("3");
      addButton("0");
      addButton(".");

      // Add clear entry button

      clearButton = new JButton("CE");
      buttonPanel.add(clearButton);

      class ClearButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            display.setText("");
         }
      }
      ActionListener listener = new ClearButtonListener();

      clearButton.addActionListener(new
            ClearButtonListener());

      add(buttonPanel, "Center");
   }

   /**
      Adds a button to the button panel.
      @param label  the button label
   */
   private void addButton(final String label)
   {
      class DigitButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            // Don't add two decimal points
            if (label.equals(".")
                  && display.getText().indexOf(".") != -1)
            {
               return;
            }

            // Append label text to button
            display.setText(display.getText() + label);
         }
      }

      JButton button = new JButton(label);
      buttonPanel.add(button);
      ActionListener listener = new DigitButtonListener();
      button.addActionListener(listener);
   }

   /**
      Gets the value that the user entered.
      @return  the value in the text field of the keypad
   */
   public double getValue()
   {
      return Double.parseDouble(display.getText());
   }

   /**
      Clears the display.
   */
   public void clear()
   {
      display.setText("");
   }
}
```

worked_example_1/ATMFrame.java

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;

/**
```

```
   A frame displaying the components of an ATM.
*/
public class ATMFrame extends JFrame
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 300;

   private JButton aButton;
   private JButton bButton;
   private JButton cButton;

   private KeyPad pad;
   private JTextArea display;

   private ATM theATM;

   /**
      Constructs the user interface of the ATM frame.
   */
   public ATMFrame(ATM anATM)
   {
      theATM = anATM;

      // Construct components
      pad = new KeyPad();

      display = new JTextArea(4, 20);

      aButton = new JButton("  A  ");
      aButton.addActionListener(new AButtonListener());

      bButton = new JButton("  B  ");
      bButton.addActionListener(new BButtonListener());

      cButton = new JButton("  C  ");
      cButton.addActionListener(new CButtonListener());

      // Add components

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(aButton);
      buttonPanel.add(bButton);
      buttonPanel.add(cButton);

      setLayout(new FlowLayout());
      add(pad);
      add(display);
      add(buttonPanel);
      showState();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   /**
      Updates display message.
   */
   public void showState()
   {
      int state = theATM.getState();
      pad.clear();
```

```
         if (state == ATM.START)
         {
            display.setText("Enter customer number\nA = OK");
         }
         else if (state == ATM.PIN)
         {
            display.setText("Enter PIN\nA = OK");
         }
         else if (state == ATM.ACCOUNT)
         {
            display.setText("Select Account\n"
                  + "A = Checking\nB = Savings\nC = Exit");
         }
         else if (state == ATM.TRANSACT)
         {
            display.setText("Balance = "
                  + theATM.getBalance()
                  + "\nEnter amount and select transaction\n"
                  + "A = Withdraw\nB = Deposit\nC = Cancel");
         }
      }

      class AButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            int state = theATM.getState();
            if (state == ATM.START)
            {
               theATM.setCustomerNumber((int) pad.getValue());
            }
            else if (state == ATM.PIN)
            {
               theATM.selectCustomer((int) pad.getValue());
            }
            else if (state == ATM.ACCOUNT)
            {
               theATM.selectAccount(ATM.CHECKING);
            }
            else if (state == ATM.TRANSACT)
            {
               theATM.withdraw(pad.getValue());
               theATM.back();
            }
            showState();
         }
      }

      class BButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            int state = theATM.getState();
            if (state == ATM.ACCOUNT)
            {
               theATM.selectAccount(ATM.SAVINGS);
            }
            else if (state == ATM.TRANSACT)
            {
               theATM.deposit(pad.getValue());
               theATM.back();
            }
            showState();
         }
      }

      class CButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            int state = theATM.getState();
            if (state == ATM.ACCOUNT)
            {
               theATM.reset();
            }
            else if (state == ATM.TRANSACT)
            {
               theATM.back();
            }
            showState();
         }
      }
   }
```

worked_example_1/ATMViewer.java

```
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
   A graphical simulation of an automatic teller machine.
*/
public class ATMViewer
{
   public static void main(String[] args)
   {
      ATM theATM;

      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(null, "Error opening accounts file.");
         return;
      }

      JFrame frame = new ATMFrame(theATM);
      frame.setTitle("First National Bank of Java");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```