CHAPTER 9

A Case Study—A Card Game

In this third case study we will examine a simple card game, a version of solitaire. A slightly different rendition of this program was presented in C++ in the first edition of this book and rewritten to use the MFC library in another book [Budd 1999]. The program was translated into Java in the second edition and revised once again in Java in yet another book [Budd 1998b]. The program presented here is one more revision, this time translated into C#.

I have used this case study in so many different forms because the development of this program is a good illustration of the power of inheritance and overriding. We will get to those aspects after first considering some of the basic elements of the game. The complete source for the program can be viewed in Appendix C.

9.1 ⊡ The Class PlayingCard

Wherever possible, software development should strive for the creation of general purpose reusable classes, classes that make minimal demands on their environment and hence can be carried from one application to another. This idea is illustrated by the first class, which represents a playing card. The class defining the playing card abstraction is shown in Figure 9.1. We have examined aspects of this class in earlier chapters.

The methods isFaceUp, rank, suit, and color have been writen as *properties*. Since they include only a get clause and no set feature, they are properties that can be read and not modified. Two enumerated data types are used by the playing card class. The enumerated type Color is provided by the standard run-time system. The class Suits is specific to this project and is defined as follows.

```
public class PlayingCard
{
  public PlayingCard (Suits sv, int rv)
    { s = sv; r = rv; faceUp = false; }

  public bool isFaceUp
  {
    get { return faceUp; }
  }

  public void flip ()
  {
    faceUp = ! faceUp;
  }

  public int rank
  {
    get { return r; }
  }

  public Suits suit
  {
    get { return s; }
  }

  public Color color
  {
    get
    {
      if ( suit == Suits.Heart || suit == Suits.Diamond )
        { return Color.Red; }
      return Color.Black;
    }
  }

  private bool faceUp;
  private int r;
  private Suits s;
}
```

⊡ Figure 9.1 — The definition of the class PlayingCard


```
public enum Suits { Spade, Diamond, Club, Heart };
```

In C#, unlike C++, enumerated constants must be prefixed by their type name. You can see this in the method color through the use of names such as Color.Black or Suits.Heart, instead of simply Black or Heart.

The class PlayingCard has no information about the application in which it is developed and can easily be moved from this program to another program that uses the playing card abstraction.

9.2 ⊡ Data and View Classes

Techniques used in the creation of visual interfaces have undergone frequent revisions, and this trend will likely continue for the foreseeable future. For this reason it is useful to separate classes that contain data values, such as the PlayingCard abstraction, from classes that are used to provide a graphical display of those values. By doing so the display classes can be modified or replaced as necessary, leaving the original data classes untouched.

The display of the card abstraction will be provided by the class CardView. To isolate the library-specific aspects of the card view, the actual display method is declared as *abstract* (see Section 8.5). This will later be subclassed and replaced with a function that will use the C# graphics facilities to generate the graphical interface.

```
public abstract class CardView
{
  public abstract void display (PlayingCard aCard, int x, int y);

  public static int Width = 50;
  public static int Height = 70;
}
```

By not only separating playing cards from card views but also separating the concept of a card view from a specific implementation, we isolate any code that is specific to a single graphics library. The effect is that the majority of the code in the application has no knowledge of the graphics library being used. This facilitates any future modifications to the graphics aspects of the application, which are the features most likely to change.

The CardView class encapsulates a pair of static constants that represent the height and width of a card on the display. The PlayingCard class itself knows nothing about how it is displayed. One abstract method is prototyped. This method will display the face of a card at a given position on the display.

It can be argued that even including the height and width as values in this class is introducing some platform dependencies, but these are less likely to change than are the libraries used to perform the actual graphical display.

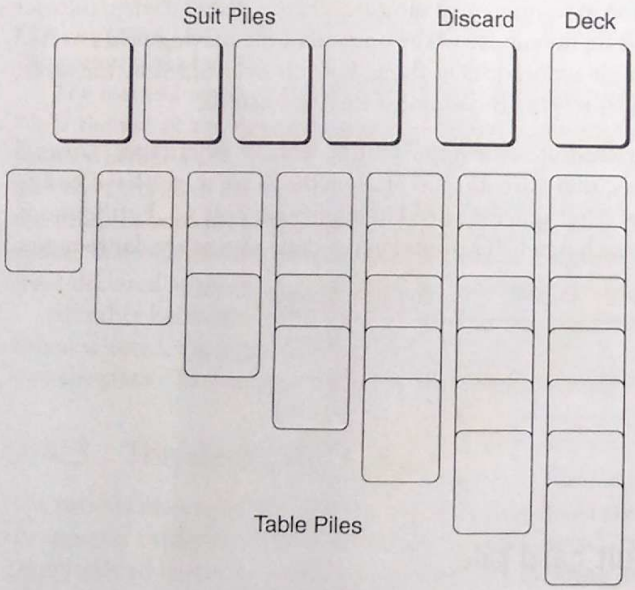
9.3 ⊡ The Game

The version of solitaire we will describe is known as klondike. The countless variations on this game make it probably the most common version of solitaire, so much so that when you say "solitaire," most people think of klondike. The version we will use is the one described in [Morehead 1949]; variations on the basic game are numerous.

The layout of the game is shown in Figure 9.2. A single standard pack of 52 cards is used. The *tableau*, or playing table, consists of 28 cards in 7 piles. the first pile has 1 card, the second 2, and so on up to 7. The top card of each pile is initially faceup; all other cards are facedown.

The suit piles (sometimes called *foundations*) are built up from aces to kings in suits. They are constructed above the tableau as the cards become available. The object of the game is to build all 52 cards into the suit piles.

The cards that are not part of the tableau are initially all in the *deck*. Cards in the deck are facedown and are drawn one by one from the deck and placed



⊡ Figure 9.2 — Layout for the solitaire game

faceup on the *discard pile*. From there they can be moved onto either a tableau pile or a foundation. Cards are drawn from the deck until the pile is empty; at this point, the game is over if no further moves can be made.

Cards can be placed on a tableau pile only on a card of next-higher rank and opposite color. They can be placed on a foundation only if they are the same suit and the next-higher card or if the foundation is empty and the card is an ace. Spaces in the tableau that arise during play can be filled only by kings.

The topmost card of each tableau pile and the topmost card of the discard pile are always available for play. The only time more than one card is moved is when an entire collection of faceup cards from a tableau (called a *build*) is moved to another tableau pile. This can be done if the bottommost card of the build can be legally played on the topmost card of the destination. Our initial game will not support the transfer of a build, but we will discuss this as a possible extension. The topmost card of a tableau is always faceup. If a card is moved from a tableau, leaving a facedown card on the top, the latter card can be turned faceup.

From this short description, it is clear that the game of solitaire mostly involves manipulating piles of cards. Each type of pile has many features in common with the others and a few aspects unique to the particular type. In the next section, we will investigate in detail how inheritance can be used in such circumstances to simplify the implementation of the various card piles by providing a common base for the generic actions and permitting this base to be redefined when necessary.

9.4 ⊡ Card Piles—Inheritance in Action

Much of the behavior we associate with a card pile is common to each variety of pile in the game. For example, each pile maintains a collection containing the cards in the pile, and the operations of inserting and deleting elements from this collection are common. Other operations are given default behavior in the class CardPile, but they are sometimes overridden in the various subclasses. The class CardPile is shown in Figure 9.3.

Each card pile maintains the coordinate location for the upper-left corner of the pile, as well as a collection that contains the card in the pile. The Stack abstraction from the standard run-time library is used to hold the cards. All these values are set by the constructor for the class. The data fields, located near the end of the declaration, are declared as protected and are thus accessible to member functions associated with this class and to member functions associated with subclasses.

The three functions top(), pop(), and isEmpty() manipulate the list of cards using functions provided by the Stack class. The remaining five operations defined in class CardPile are common to the abstract notion of our card piles, but they differ in details in each case. For example, the function canTake(PlayingCard) asks whether it is legal to place a card on the given pile. A card can be added to a foundation

```
public class CardPile {
  public CardPile (int xl, int yl )
    { x = xl; y = yl; pile = new Stack(); }

  public PlayingCard top
    { get { return (PlayingCard) pile.Peek (); } }

  public bool isEmpty
    { get { return pile.Count == 0; }  }

  public PlayingCard pop
    { get { return (PlayingCard) pile.Pop (); } }

    // the following are sometimes overridden
  public virtual bool includes (int tx, int ty ) {
    return( ( x <= tx ) && ( tx <= x + CardView.Width ) &&
        ( y <= ty ) && ( ty <= y + CardView.Height ) );
  }

  public virtual void select (int tx, int ty ) {
    // do nothing–override
  }

  public virtual void addCard (PlayingCard aCard )
    { pile.Push(aCard); }

  public virtual void display (CardView cv) {
    if ( isEmpty ) {
      cv.display(null, x, y);
    } else {
      cv.display((PlayingCard) pile.Peek(), x, y );
    }
  }

  public virtual bool canTake (PlayingCard aCard)
    { return false; }

  protected int x, y; // coordinates of the card pile
  protected Stack pile; // card pile data
}
```

▫ Figure 9.3 — Description of the class CardPile

pile, for instance, only if it is an ace and the foundation is empty or if the card is the same suit as the current topmost card in the pile and has the next-higher value. A card can be added to a tableau pile, on the other hand, only if the pile is empty and the card is a king or if it is of the opposite color as the current topmost card in the pile and has the next-lower value.

The actions of the five virtual functions defined in CardPile can be characterized as follows.

includes Determines if the coordinates given as arguments are contained within the boundaries of the pile. The default action simply tests the topmost card; this is overridden in the tableau piles to test all card values.

canTake Tells whether a pile can take a specific card. Only the tableau and suit piles can take cards, so the default action is simply to return no; this is overridden in the two classes mentioned.

addCard Adds a card to the card list. It is redefined in the discard pile class to ensure that the card is faceup.

display Displays the card deck. The default method merely displays the topmost card of the pile but is overridden in the tableau class to display a column of cards. The top half of each hidden card is displayed. So that the playing surface area is conserved, only the topmost and bottommost faceup cards are displayed (this permits us to give definite bounds to the playing surface).

select Performs an action in response to a mouse click. It is invoked when the user selects a pile by clicking the mouse in the portion of the playing field covered by the pile. The default action does nothing, but it is overridden by the table, deck, and discard piles to play the topmost card, if possible.

The following table illustrates the important benefits of inheritance. Given 5 operations and 5 classes, there are 25 potential methods we might have had to define. By making use of inheritance we need to implement only 13. Furthermore, we are guaranteed that each pile will respond in the same way to similar requests.

	CardPile	SuitPile	DeckPile	DiscardPile	TableauPile
includes	×				×
canTake	×	×			×
addCard	×			×	
display	×				×
select	×		×	×	×

9.4.1 The default card pile

We will examine each of the subclasses of CardPile in detail, pointing out various uses of object-oriented features. Each of the five virtual methods is first defined

in the class CardPile. These implementations will represent the default behavior should they not be overridden. The implementation of these methods was shown in Figure 9.3.

9.4.2 The suit piles

The simplest subclass is the class SuitPile, which represents the pile of cards at the top of the playing surface. This is the pile being built up in suit from ace to king. The implementation of this class is as follows.

```
public class SuitPile : CardPile {
  public SuitPile (int x, int y) : base(x, y) {  }

  public override bool canTake (PlayingCard aCard ) {
    if( isEmpty )
      { return( aCard.rank == 0 ); }
    PlayingCard topCard = top;
    return( ( aCard.suit == topCard.suit ) &&
       ( aCard.rank == topCard.rank + 1 ) );
  }
}
```

The class SuitPile defines only two methods. The constructor for the class takes two integer arguments and does nothing more than invoke the constructor for the parent class CardPile.

The method canTake overrides the similarly named method in the parent class. Note the use of the keyword override that indicates this fact. This method determines whether a card can be placed on the pile. A card is legal if the pile is empty and the card is an ace (that is, has rank zero) or if the card is the same suit as the topmost card in the pile and of the next-higher rank (for example, a three of spades can only be played on a two of spades). Since the methods rank and suit were declared as properties, they can be invoked without parentheses.

All other behavior of the suit pile is the same as that of our generic card pile. When selected, a suit pile does nothing. When a card is added, it is simply inserted into the stack. To display the pile only the topmost card is drawn.

9.4.3 The deck pile

The DeckPile maintains the deck from which new cards are drawn. It differs from the generic card pile in two ways. When constructed, rather than creating an empty pile of cards, it initializes itself by first creating an array containing the 52 cards in a conventional deck, then randomly selecting elements from this collection to generate a sorted deck. The method select is invoked when the mouse

button is used to select the card deck. If the deck is empty, it does nothing. Otherwise, the topmost card is removed from the deck and added to the discard pile.

```
public class DeckPile : CardPile {
  public DeckPile (int x, int y) : base(x, y) {
    // create the new deck
    // first put cards into a local array
    ArrayList aList = new ArrayList ();
    for( int i = 0; i <= 12; i++) {
      aList.Add(new PlayingCard(Suits.Heart, i));
      aList.Add(new PlayingCard(Suits.Diamond, i));
      aList.Add(new PlayingCard(Suits.Spade, i));
      aList.Add(new PlayingCard(Suits.Club, i));
    }
      // then pull them out randomly
    Random myRandom = new Random( );
    for(int count = 0; count < 52; count++) {
      int index = myRandom.Next(aList.Count);
      addCard( (PlayingCard) aList [index] );
      aList.RemoveAt(index);
    }
  }

  public override void select (int tx, int ty) {
    if ( isEmpty ) { return; }
    Game.discardPile().addCard( pop );
  }
}
```

The implementation of the select method presents us with a new problem. When the mouse is pressed on the deck pile, the desired action is to move a card from the deck pile on to the discard pile, turning it faceup in the process. The problem is that we now need to refer to a single unique card pile—namely, the pile that represents the discard pile.

One approach would be to define the various card piles as global variables, which then could be universally accessed. In fact, this approach is used in the program described in my earlier C++ version of the game in the first edition of this book. But many languages, such as Java and C#, do not have global variables. There is good reason for this. Global variables tend to make it difficult to understand the flow of information through a program, since they can be accessed from any location (that's what makes them global).

A better and more object-oriented alternative to the use of global variables is a series of static values. This reduces the number of global values to one: the class name. Static methods in the class can then be used to access further states. In our program we will name this class Game. A discussion of the details of this class will be postponed until after the description of the various card piles.

9.4.4 The discard pile

The class DiscardPile redefines the addCard and select methods. The class is described as follows.

```
public class DiscardPile : CardPile {
  public DiscardPile (int x, int y ) : base(x, y) { }

  public override void addCard (PlayingCard aCard) {
    if( ! aCard.isFaceUp )
      { aCard.flip(); }
    base.addCard( aCard );
  }

  public override void select (int tx, int ty) {
    if( isEmpty ) { return; }
    PlayingCard topCard = pop;
    for( int i = 0; i < 4; i++ ) {
      if( Game.suitPile(i).canTake( topCard ) ) {
        Game.suitPile(i).addCard( topCard );
        return;
      }
    }

    for( int i = 0; i < 7; i++ ) {
      if( Game.tableau(i).canTake( topCard ) ) {
        Game.tableau(i).addCard( topCard );
        return;
      }
    }
    // nobody can use it, put it back on our stack
    addCard(topCard);
  }
}
```

The implementation of these methods is interesting in that they exhibit two very different forms of inheritance. The select method *overrides* or *replaces* the default behavior provided by class CardPile, replacing it with code that when invoked (when the mouse is pressed over the card pile) checks to see if the topmost card can be played on any suit pile or, alternatively, on any tableau pile. If the card cannot be played, it is kept in the discard pile.

The method addCard is a different sort of overriding. Here the behavior is a *refinement* of the default behavior in the parent class. That is, the behavior of the parent class is completely executed, and in addition, new behavior is added. In this case, the new behavior ensures that when a card is placed on the discard pile it is always faceup. After satisfying this condition, the code in the parent class is invoked to add the card to the pile. The keyword base is necessary to avoid the confusion with the addCard method being defined. In Java the same problem would be addressed by sending a message to super (as in super.addCard(aCard)).

Another form of refinement occurs in the constructors for the various subclasses. Each must invoke the constructor for the parent class to guarantee that the parent is properly initialized before the constructor performs its own actions. The parent constructor is invoked by an initializer clause inside the constructor for the child class.

9.4.5 The tableau piles

The most complex of the subclasses of CardPile is that used to hold a tableau, or table pile. The implementation of this class redefines nearly all of the virtual methods defined in ClassPile. When initialized by the constructor, the tableau pile removes a certain number of cards from the deck, placing them in its own pile. The number of cards so removed is determined by an additional argument to the constructor. The topmost card of this pile is then displayed faceup.

```
public class TablePile : CardPile {
  public TablePile (int x, int y, int c) : base(x, y) {
    // initialize our pile of cards
    for(int i = 0; i < c; i++ ) {
      addCard(Game.deckPile().pop);
    }
    top.flip();
  }

  public override bool canTake (PlayingCard aCard ) {
    if( isEmpty ) { return(aCard.rank == 12); }
    PlayingCard topCard = top;
    return( ( aCard.color != topCard.color ) &&
      ( aCard.rank    == topCard.rank - 1 ) );
  }
```

```
    public override bool includes (int tx, int ty) {
      return( ( x <= tx ) && ( tx <= x + CardView.Width ) &&
          ( y <= ty ) );
    }

    public override void select (int tx, int ty) {
      if( isEmpty ) { return; }
      // if face down, then flip
      PlayingCard topCard = top;
      if( ! topCard.isFaceUp ) {
        topCard.flip();
        return;
      }
      // else see if any suit pile can take card
      topCard = pop;
      for(int i = 0; i < 4; i++ ) {
        if( Game.suitPile(i).canTake( topCard ) ) {
          Game.suitPile(i).addCard( topCard );
          return;
        }
      }
      // else see if any other table pile can take card
      for(int i = 0; i < 7; i++ ) {
        if( Game.tableau(i).canTake( topCard ) ) {
          Game.tableau(i).addCard( topCard );
          return;
        }
      }
      addCard( topCard );
    }

    public override void display (CardView cv) {
      Object [ ] cardArray = pile.ToArray();
      int size = pile.Count;
      int hs = CardView.Height / 2; // half size
      int ty = y;
      for (int i = pile.Count - 1; i >= 0; i--) {
        cv.display((PlayingCard) cardArray[i], x, ty);
        ty += hs;
      }
    }
}
```


A card can be added to the pile (method canTake) only if the pile is empty and the card is a king, or if the card is the opposite color from that of the current topmost card and one smaller in rank. When a mouse press is tested to determine if it covers the pile (method includes), the bottom bound is not tested, since the pile may be of variable length. When the pile is selected, the topmost card is flipped if it is facedown. If it is faceup, an attempt is made to move the card first to any available suit pile and then to any available table pile. Only if no pile can take the card is it left in place. Finally, to display a pile, all the underlying cards are displayed. The stack must be converted into an array to do this since we must access the cards top to bottom, which is the opposite of the order that stack elements would normally be enumerated.

9.5 ⊡ Playing the Polymorphic Game

The need for the class Game was described earlier. This class holds the actual card piles used by the program, making them available through methods that are declared as static. Because these methods are static, they can be accessed using only the class name as a basis.

The definition of this class is shown in Figure 9.4. The game manager stores the various card piles in an array, one that is declared as CardPile, although the values are polymorphic and hold a variety of different types of card piles. These values are initialized in the constructor, which is declared as static. A static constructor will be executed when the program begins execution.

By storing the card values in a polymorphic array, the game manager need not distinguish the characteristics of the individual piles. For example, to repaint the display it is only necessary to tell each pile to repaint itself. The method display will be different, depending on the actual type of card pile. Similarly, to respond to a mouse down, the manager simply cycles through the list of card piles.

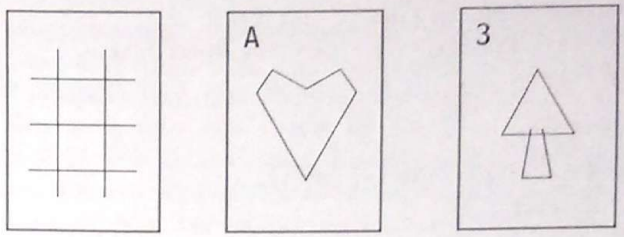
9.6 ⊡ The Graphical User Interface

We have taken pains in the development of this program to isolate the details of both the graphical user interface and of the high-level program execution. This is because of all the elements of a program, the user interface is the most likely to require change as new graphical libraries are introduced or existing libraries are changed. Similarly, the way that applications are initiated using C# introduces details that would have obscured the overall design of the application.

The card images are simple line drawings. Diamonds and hearts are drawn in red, spades and clubs in black. The hash marks on the back are drawn in yellow.

```
public class Game {
  static Game () {
    allPiles = new CardPile[ 13 ];
    allPiles[0] = new DeckPile(335, 5 );
    allPiles[1] = new DiscardPile(268, 5 );
    for( int i = 0; i < 4; i++ ) {
      allPiles[2 + i] = new SuitPile(15 + 60 * i, 5);
    }
    for( int i = 0; i < 7; i++ ) {
      allPiles[6+i] = new TablePile(5+55*i, 80, i+1);
    }
  }

  public static void paint (CardView cv) {
    for( int i = 0; i < 13; i++ ) {
      allPiles[i].display(cv );
    }
  }

  public static void mouseDown (int x, int y) {
    for( int i = 0; i < 13; i++ ) {
      if( allPiles[i].includes(x, y) ) {
        allPiles [i].select(x, y);
      }
    }
  }

  public static CardPile deckPile ()
    { return allPiles[0]; }

  public static CardPile discardPile ()
    { return allPiles[1]; }

  public static CardPile tableau (int index)
    { return allPiles[6+index]; }

  public static CardPile suitPile (int index)
    { return allPiles[2+index]; }

  private static CardPile[] allPiles;
}
```

⊡ Figure 9.4 — The class Game



We deal first with the user interface. Recall that the display of a card was provided by a method CardView that was described as abstract. To produce actual output, we must create a subclass that implements the pure virtual methods. This class we will call WinFormsCardView:

```
public class WinFormsCardView : CardView {
  public WinFormsCardView (Graphics aGraphicsObject) {
    g = aGraphicsObject;
  }

  public override void display (PlayingCard aCard,int x,int y) {
    if  (aCard == null) {
      Pen myPen = new Pen(Color.Black,2);
      Brush myBrush = new SolidBrush (Color.White);
      g.FillRectangle(myBrush,x,y,CardView.Width,CardView.Height);
      g.DrawRectangle(myPen,x,y,CardView.Width,CardView.Height);
    } else {
      paintCard (aCard,x,y);
    }
  }

  private void paintCard (PlayingCard aCard,int x,int y) {
    String [] names = { "A","2","3","4","5",
      "6","7","8","9","10","J","Q","K" };

    Pen myPen = new Pen (Color.Black,2);
    Brush myBrush = new SolidBrush (Color.White);

    g.FillRectangle (myBrush,x,y,CardView.Width,CardView.Height);
    g.DrawRectangle(myPen,x,y,CardView.Width,CardView.Height);
    myPen.Dispose();
    myBrush.Dispose();

    // draw body of card with a new pen-color
```

```
    if (aCard.isFaceUp) {
      if (aCard.color == Color.Red) {
        myPen = new Pen  (Color.Red,1);
        myBrush = new SolidBrush (Color.Red);
      } else {
        myPen = new Pen  (Color.Blue,1);
        myBrush = new SolidBrush (Color.Blue);
      }
      g.DrawString (names[ aCard.rank ],
        new Font("Times New Roman",10),myBrush,x+3,y+7);
      if (aCard.suit == Suits.Heart) {
        g:DrawLine(myPen,x+25,y+30,x+35,y+20);
        g.DrawLine(myPen,x+35,y+20,x+45,y+30);
        g.DrawLine(myPen,x+45,y+30,x+25,y+60);
        g.DrawLine(myPen,x+25,y+60,x+5,y+30);
        g.DrawLine(myPen,x+5,y+30,x+15,y+20);
        g.DrawLine(myPen,x+15,y+20,x+25,y+30);
      } else if (aCard.suit == Suits.Spade) {
        .
        .   see code in appendix
        .
      } else if (aCard.suit == Suits.Diamond) {
        .
        .
        .
      } else if (aCard.suit == Suits.Club) {
        .
        .
        .
      }
    } else {   // face down
      myPen = new Pen (Color.Green,1);
      myBrush = new SolidBrush  (Color.Green);
      .
      .
      .
    }
  }
  private Graphics g;
}
```

This is not a text on graphics, so the actual display will be rather simple. Basically, a card draws itself as a rectangle with a textual description. Empty piles are drawn in green, the backsides of cards in yellow, the faces in the appropriate color.

Graphical output in the C# library is based around a type of object from class Graphics. This object is passed as constructor to the class and stored in the variable

g. Details of the graphical output routines provided by the Windows library will not be discussed here, although many of the names are self-explanatory. The display for our game is rather primitive, consisting simply of line rectangles and the textual display of card information.

Applications in the C# framework are created by subclassing from a system-provided class named System.WinForms.Form and overriding certain key methods. Much of the structure of the class is generated automatically if one uses a development environment, such as the Studio application. In the following we have marked the generated code with comments. The programmer then edits this code to fit the specific application. The final class is as follows.

```
public class Solitaire : System.WinForms.Form {
    // start of automatically generated code
  private System.ComponentModel.Container components;

  public Solitaire() {
    InitializeComponent();
  }

  public override void Dispose() {
    base.Dispose();
    components.Dispose();
  }

  private void InitializeComponent() {
    this.components = new System.ComponentModel.Container ();
    this.Text = "Solitaire";
    this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
    this.ClientSize = new System.Drawing.Size (392, 373);
  }
    // end of automatically generated code

  protected override void OnMouseDown (MouseEventArgs e ) {
    Game.mouseDown(e.X, e.Y);
    this.Invalidate(); // force screen redraw
  }

  protected override void OnPaint (PaintEventArgs pe ) {
    Graphics g = pe.Graphics;
    CardView cv = new WinFormsCardView(g);
    Game.paint(cv);
  }
```

```
    public static void Main(string[] args)
      { Application.Run(new Solitaire()); }
}
```

The window class is responsible for trapping the actual mouse presses and repainting the window. In our application these activities are simply passed on to the game manager. As with Java, execution begins with the method named Main. This method invokes a static method from a system class named Application, passing it an instance of the game controller class.

Summary ⊡

The solitaire game is a standard example program found in many textbooks. We have here used the program as a case study to illustrate a number of important concepts. In the design of the PlayingCard and CardView classes, we have separated a model from a view. This is important, since aspects of the view are likely to change more rapidly than aspects of the model. Extending this further, we have defined the view as an abstract class and thereby hidden all Windows-specific features in an implementation of this class. Moving to a different graphical library would therefore simply involve changing the implementation of this abstract class.

Probably the most notable feature of the game is the use of inheritance and overriding, exemplified by the classes CardPile and its various subclasses. Through the use of overriding we avoid having to write a large amount of code. Furthermore, the use of a polymorphic variable to reference the various classes simplifies the task of redrawing the screen or handling mouse operations.

Further Reading ⊡

Source for the various earlier versions of this program can be found on my Web site, `http://www.cs.orst.edu/~budd`.

We have in this simple application only scratched the surface of the functionality provided by the C# system. However, the details of how Windows programs are created are complicated and beyond the issues being discussed here. A good introduction to the C# system is provided by Gunnerson [Gunnerson 2000].

Self-Study Questions ⊡

1. Why should the class PlayingCard be written so as to have no knowledge of the application in which it is being used?

2. Why is it useful to separate the class PlayingCard from the class that will draw the image of the playing card in the current application?
3. Why is it further useful to define the interface for CardView as an abstract class and then later supply an implementation of this class that uses the C# graphics facilities?
4. What are the different types of card piles in this solitare game?
5. What methods in CardPile are potentially overridden? What methods are not overridden? How can you tell from the class description which are which?
6. In what way does the variable allPiles exhibit polymorphism?
7. How does the polymorphism in allPiles simplify the design of the program?

Exercises ⊡

1. The solitaire game has been designed to be as simple as possible. A few features are somewhat annoying but can be easily remedied with more coding. These include the following.
   a. The topmost card of a tableau pile should not be moved to another tableau pile if there is another faceup card below it.
   b. An entire build should not be moved if the bottommost card is a king and there are no remaining facedown cards.

   For each, describe what procedures need to be changed, and give the code for the updated routine.
2. The following are common variations of klondike. For each, describe which portions of the solitaire program need to be altered to incorporate the change.
   a. If the user clicks on an empty deck pile, the discard pile is moved (perhaps with shuffling) back to the deck pile. Thus, the user can traverse the deck pile multiple times.
   b. Cards can be moved from the suit pile back into the tableau pile.
   c. Cards are drawn from the deck three at a time and placed on the discard pile in reverse order. As before, only the topmost card of the discard pile is available for playing. If fewer than three cards remain in the deck pile, all the remaining cards (as many as that may be) are moved to the discard pile. (In practice, this variation is often accompanied by variation a, permitting multiple passes through the deck.)
   d. The same as variation c, but any of the three selected cards can be played. (This requires a slight change to the layout as well as an extensive change to the discard pile class.)
   e. Any royalty card, not simply a king, can be moved onto an empty tableau pile.

3. The game "thumb and pouch" is similar to klondike except that a card may be built on any card of next-higher rank, of any suit but its own. Thus, a nine of spades can be played on a ten of clubs but not on a ten of spades. This variation greatly improves the chances of winning. (According to Morehead [Morehead 1949], the chances of winning Klondike are 1 in 30, whereas the chances of winning thumb and pouch are 1 in 4.) Describe what portions of the program need to be changed to accommodate this variation.

CHAPTER 10

Subclasses and Subtypes

There is a paradox that lies at the heart of the way inheritance and substitution are used in statically typed object-oriented languages. This paradox derives from the twin concepts of *subclass* and *subtype*. In this chapter we will explore these concepts and this paradox.

To say that one class is a *subclass* of another is to simply assert that it has been built using inheritance. The new class is declared using an existing class as a basis, as in the following Java class declaration.

```
class Child extends Parent {
   ... // class definition
}
```

The point is that the subclass relationship is asserting a statement about definition, about how the new class was constructed. It says nothing about the meaning or purpose of the child class.

10.1 ⊡ Substitutability

One of the more interesting features of statically typed object-oriented languages is that the type associated with a *value* held by a variable may not exactly match the type associated with the *declaration* for that variable. We saw this near the end of the billiard simulation program in Chapter 7, where a variable declared as a GraphicalObject in fact held a value of type Ball, Wall, or Hole. To appreciate how unusual this is, note that variables in conventional typed programming languages never have this property. A variable declared as a Integer, for example, can never hold a value of type String.