



## CHAPTER 3

---

# *Case study: a video poker machine*

- |     |                             |    |     |  |    |
|-----|-----------------------------|----|-----|--|----|
| 3.1 | Playing video poker         | 56 | 3.5 | ComPok: a COM-based poker game         | 70 |
| 3.2 | The Poker.Card class        | 58 | 3.6 | IEPok: an Internet Explorer poker game | 73 |
| 3.3 | The Poker.Hand class        | 61 | 3.7 | Designing a complete game              | 76 |
| 3.4 | SimPok: a simple poker game | 68 | 3.8 | Summary                                | 77 |

In chapter 1, we considered the architecture of a loan management system for a bank. In doing so, we saw how .NET gives us the building blocks we need to create multi-tier applications with shared back-end systems and multiple interfaces to customers, business partners, and other bank departments. In this chapter, we begin a case study in which we employ these building blocks to implement a 3-tier, multiuser, distributed video poker machine.

I've chosen video poker for several reasons:

- Unlike examples from mainstream business or commerce, the rules of poker can be fully explained in a few paragraphs, leaving us free to concentrate on .NET development.

- A video poker machine can have all the important ingredients of a distributed, client/server, multi-interface application. It can be naturally implemented as a game engine and a set of interfaces. We'll reuse the engine to implement different game applications based on COM, Internet Explorer, remoting, Windows services, message queuing, XML Web services, Windows Forms, Web Forms, and Mobile Forms. We'll also use SQL Server and ADO.NET to store play history and provide data to drive the payout management algorithm.
- Video poker is a real-world application which generates millions of dollars in profits each year in places like Las Vegas and Atlantic City.
- Best of all, we'll need to play lots of poker to test our work.

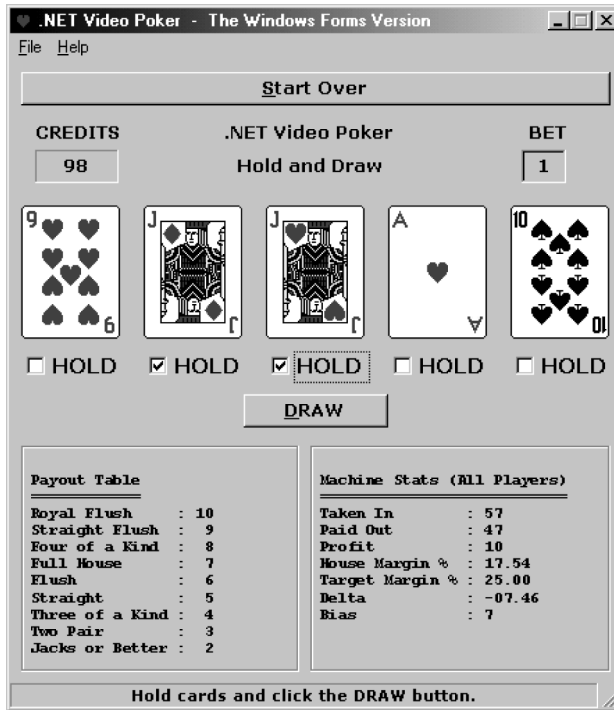
We'll develop the poker engine, a DLL assembly, in this and the next chapter. In later chapters, we'll develop different interfaces to the engine using most of the important features of .NET. In doing so, we'll get a reasonably complete tour of .NET application development. In the meantime, we take a short break from .NET and play a little poker.

### 3.1 **PLAYING VIDEO POKER**

Video poker took the casino industry by storm in the early 1980s. In Las Vegas casinos, and elsewhere, it now accounts for a greater share of income than traditional slot machines. The game is simple. You play against a machine which acts as dealer and you insert money to receive credits. Then you make a bet and hit the Deal button. The machine displays five cards from the deck. The idea is to make the best possible poker hand out of these five cards by holding onto the best cards and drawing replacements for those you wish to discard. You do this by selecting the cards you want to hold and clicking Draw. The cards you hold are retained while the others are replaced. At this point the hand is scored and you either win or lose. Typically to win you need a pair of jacks or better. If you win, your winnings are calculated by multiplying the score for the hand by the amount of your bet. The total is added to your existing credits.

Figure 3.1 shows the Windows Forms version of the game. The screenshot shows the state of the game after we have placed our bet and received five cards. At this stage the bet text box in the top right of the screen is disabled to prevent us from changing our bet mid-game. In the top left we can see that we have 98 credits remaining. Underneath the two jacks we have checked the checkboxes to indicate that we want to hold onto these two cards when we draw. By doing so, we are guaranteed to win since the minimum scoring hand is a pair of jacks or better.

At this point we would click DRAW and the three remaining cards would be replaced. Drawing a third jack would result in a scoring hand known as *three of a kind*.



**Figure 3.1**  
The Windows Forms version  
of video poker

### 3.1.1 Winning poker hands

Table 3.1 lists the winning poker hands, their scores, and some examples. We use a two-character identifier for a card's name. For example, the ace of diamonds has the name "AD", and so forth. This gives us a user-friendly shorthand for displaying cards on the console.

**Table 3.1** Winning video poker hands

Hand	Example	Score	Description
Royal Flush	TD JD QD KD AD	10	A straight to the ace in the same suit
Straight Flush	3H 4H 5H 6H 7H	9	A straight in the same suit
Four of a Kind	5C 5D 5H 5S QH	8	Four cards of the same number
Full House	KC KH KD 8C 8S	7	Three of a kind with any pair
Flush	9S 3S QS TS AS	6	5 cards of the same suit
Straight	8C 9S TC JC QH	5	5 cards with consecutive numbers
Three of a Kind	TD 4C 4S 3S 4D	4	Three cards of the same number
Two Pair	AD QH QD 7C 7D	3	Any pair with any pair
Jacks or Better	KD 8C 7D KS 5C	2	A pair of jacks, queens, kings, or aces

If a player bets 5 and scores three of a kind, the player's credits increase by 20 (5 times 4). Note that a card's number is the number on the card. The non-numbered cards, jack, queen, king, and ace are given the numbers, 11, 12, 13, and 14 respectively. (An ace can also double as the bottom card in a straight to the 5; e.g., AC 2D 3D 4C 5H.)

### **3.1.2 A profitable video poker machine**

In the bottom right of the screenshot in figure 3.1, we can see statistics for the machine. These statistics are based on data collected at the end of each game and stored in SQL Server. The figures include the total amounts taken in and paid out by the machine. The differences between these two figures is the profit. The house margin is the percentage profit, while the target margin represents the machine's target profit. These figures reflect data for all games played by all players and provide an overview of profitability from the machine's perspective and not from the current player's point of view.

You may be wondering about target profit. How can we set a target profit for a machine which is governed by chance? We do it by implementing a payout control algorithm which continually adjusts the odds in an effort to keep the machine on target. In figure 3.1, the machine has a 25% profit goal.

The delta figure is the difference between the target margin and the house margin. If delta is positive, then bias is zero. In other words, if the machine is meeting or exceeding its profit target, then no machine bias is necessary. Otherwise bias is calculated by taking the absolute value of delta and rounding it to an integer. The effect of this is that bias increases as the machine increasingly falls short of its profit goal. The bias value is used to tilt the odds back in favor of the machine, thus restoring profitability and reducing bias toward zero once again. In figure 3.1, the actual percentage profit is 17.54%, and the machine is falling 7.46% short of its 25% profit target. Therefore, bias is 7 and the machine will be harder to beat. The payout control algorithm is presented in detail in the following chapter.

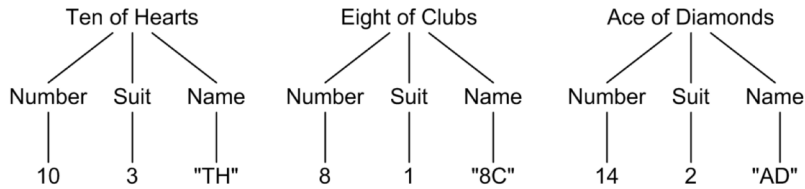
## **3.2 THE POKER.CARD CLASS**

We'll start with a simple version of the game consisting initially of just a couple of classes to represent playing cards and hands. Later in this chapter, we'll explore the design of the distributed, 3-tier version of video poker.

### **3.2.1 Designing the Card class**

It's pretty obvious that we're going to need a class to represent a playing card, and a class to represent a hand. In fact, that's all we need to get a simple game engine up and running. Figure 3.2 depicts some sample card objects.

The card class will have three public properties, `Number`, `Suit`, and `Name`. A card's number can range from 2 to 14 and its suit will be an integer ranging from 1 to 4 representing clubs, diamonds, hearts, and spades in alphabetical order. We'll use



**Figure 3.2** Example Card objects

the two-character identifier for the card's name. This simple scheme supports the efficient generation of random cards with a user-friendly way to display them as text.

### 3.2.2 Coding the Card class

Before we code the `Card` class, we observe that card objects are immutable. As in a real game, once a card is drawn from the deck, its number, suit, or name cannot be altered. Therefore, we can gain a little in both efficiency and simplicity by implementing a card's properties as public, read-only fields.

We'll need the ability to construct random playing cards to simulate dealing and drawing cards from the deck. We'll also need the ability to construct specific playing cards on demand. The code for the `Card` class is presented in listing 3.1.

#### Listing 3.1 The `Poker.Card` class

```

using System.Reflection;
[assembly: AssemblyVersion("1.0.0.0")]

namespace Poker {
    using System;

    internal class Card {
        public Card() : this(new Random()) {}

        public Card(Random r) {
            Number = r.Next(2, 15);
            Suit = r.Next(1, 5);
            Name = numberArray[Number - 2] + suitArray[Suit - 1];
        }

        public Card(string name) {
            string n = name.Substring(0, 1);
            string s = name.Substring(1, 1);
            Number = numberString.IndexOf(n) + 2;
            Suit = suitString.IndexOf(s) + 1;
            Name = name;
        }

        public readonly int Number;
        public readonly int Suit;
        public readonly string Name;

        public override string ToString() {

```

```

        return Name;
    }

    public override bool Equals(object o) {
        try {
            Card c = (Card)o;
            return c.Number == Number && c.Suit == Suit;
        } catch (Exception) {
            return false;
        }
    }

    public override int GetHashCode() {
        return (Suit<<4) + Number;
    }

    // private fields...
    private static string[] numberArray
        = {"2", "3", "4", "5", "6", "7", "8", "9", "T", "J", "Q", "K", "A"};
    private static string[] suitArray = {"C", "D", "H", "S"};
    private static string numberString = "23456789TJQKA";
    private static string suitString = "CDHS";
}
}

```

The `Card` class begins by specifying the version number of the *Poker* assembly where the `Card` class will reside. We could specify this in any of the source files which make up the assembly and choose to do so here only for convenience.

We place the code for the `Card`, and all other poker classes, inside a new namespace called `Poker`. We also specify `internal` access to the `Card` class, as it should be accessible only to a `Hand` object within the same assembly.

The `Card` class contains three constructors. The default constructor simply creates a new random number generator and calls the second constructor. To facilitate the generation of a valid pseudorandom sequence when dealing cards, we'll typically use the second constructor and pass a random number generator from the calling application. This constructor uses the private `numberArray` and `suitArray` as look up tables to create the `Name`. The third constructor accepts a card name as an argument and builds the corresponding `Card` object. This allows us to create specific cards to order when necessary.

The rest of the `Card` class should look familiar. We override `Equals` to return `true` if the two cards have the same number and suit. Therefore, we should override `GetHashCode` to ensure that cards, which are equal, hash to the same code. In this case, we combine number and suit into a single unique integer hash code. We also override `ToString` to display the card's name.

That completes the `Card` class. It provides us with a means of creating cards, displaying them on the console, comparing them for equality, and storing them in a hash table.

### 3.3 THE POKER.HAND CLASS

In video poker, cards are always assembled into a hand consisting of 5 cards. The machine starts by dealing 5 cards at random. The user can discard none, some, or all of these cards in an attempt to improve the hand's score. Therefore, a hand is an obvious choice for an application class.

#### 3.3.1 Designing the Hand class

The Hand class represents a poker hand consisting of 5 cards. Figure 3.3 depicts a Hand object for a royal flush in spades.

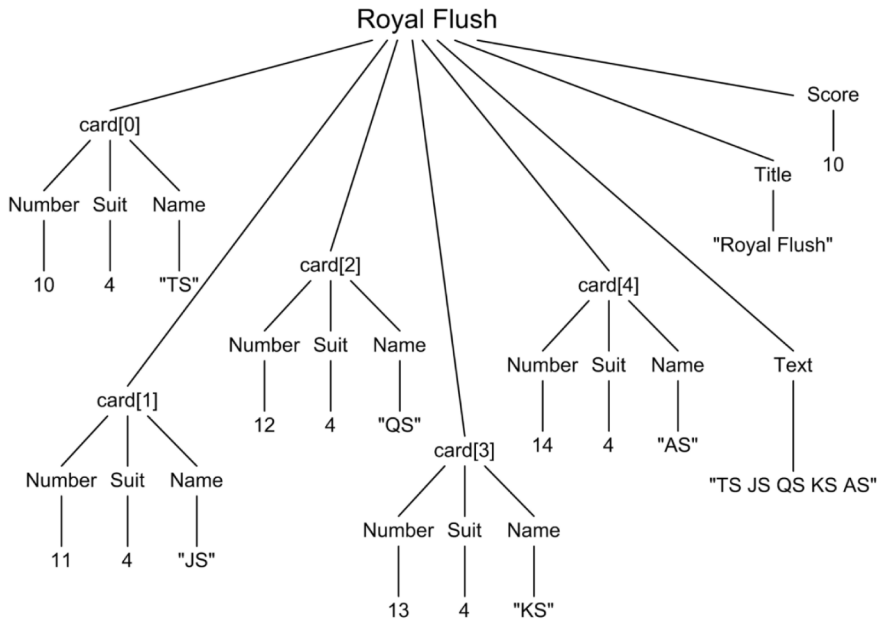


Figure 3.3 Example Hand object representing a royal flush in spades

The Hand class contains a private array of 5 card objects. It also contains public Score, Title, and Text properties. As shown in figure 3.3, for a royal flush in spades, Score is 10, Text is "TS JS QS KS KS", and Title is "Royal Flush".

#### 3.3.2 Coding the Hand class

We know that the Hand class will require one or more constructors, together with properties representing the hand's score, text, and title. The score will need to be calculated according to the scheme shown in table 3.1.

Although the Hand class is simple, calculating the hand's score takes quite a few lines of code. The full source code for the Hand class is presented in listing 3.2. In the meantime, we'll go through the code in outline here.

We begin the `Hand` class with the default constructor which simply creates a hand of 5 cards taking care to avoid duplicates:

```
public Hand() {
    Random r = new Random();
    for (int i = 0; i < 5; i++) {
        while (true) {
            cards[i] = new Card(r);
            if (containsCard(cards[i], cards, i)) continue;
            break;
        }
    }
}
```

When a player selects cards to hold, and then draws, we'll need to create a new hand from the old. To do this, we'll need a constructor that takes an existing hand, and a list of cards to hold, and creates a new hand. In fact we'll provide three different constructors to do this, as follows:

```
public Hand(string handText) {
    cardsFromString(handText);
}

public Hand(string handText, string holdString) {
    cardsFromString(handText);
    holdCards(holdString);
    draw();
}

public Hand(Hand hand, string holdString) {
    this.cards = hand.cards;
    holdCards(holdString);
    draw();
}
```

In each case the `handText` argument is a string representation of an existing hand which we get from the hand's `Text` property. These constructors support the creation of new hands from existing hands, as follows:

```
Hand newHand = Hand(oldHand.Text)
...
Hand newHand = Hand(oldHand.Text, "13") // hold 1st and 3rd cards
...
Hand newHand = Hand(oldHand, "52") // hold 5th and 2nd cards
```

Although not obvious at this point, these constructors will provide a convenient means of drawing cards in both the text-based, Windows GUI, and Web versions of the poker game.

We could implement a scheme that uses just a single hand object for each game played, as follows:

```

Hand h = new Hand();
...
h.Hold(1); // hold 1st card
h.Hold(4); // hold 4th card
...
h.Draw(); // replace 2nd, 3rd, and 5th cards

```

While this scheme would work fine, it requires maintaining the state of a single hand for the duration of a game. In contrast, we'll find that using an immutable hand object, which is discarded and replaced by a new hand when cards are drawn, provides a better model for loosely coupled, remote and Web-based versions of the game.

The rest of the `Hand` class is straightforward. Note that we only compute the `Score` property on demand:

```

public int Score { get {
    if (score < 0) calcScore();
    return score;
} }

```

We're not interested in the score until cards have been drawn and the game is over. The `calcScore`, while a little long, is simple. It simply checks for a scoring hand starting with a royal flush and ending with jacks or better.

The full `Hand` class is presented in listing 3.2.

### Listing 3.2 The `Poker.Hand` class

```

namespace Poker {
    using System;

    public class Hand {
        public Hand() {
            Random r = new Random();
            for (int i = 0; i < 5; i++) {
                while (true) {
                    cards[i] = new Card(r);
                    if (containsCard(cards[i], cards, i)) continue;
                    break;
                }
            }
        }

        public Hand(string handText) {
            cardsFromString(handText);
        }

        public Hand(string handText, string holdString) {
            cardsFromString(handText);
            holdCards(holdString);
            draw();
        }

        public Hand(Hand hand, string holdString) {

```

```

        this.cards = hand.cards;
        holdCards(holdString);
        draw();
    }

    public int Score { get {
        if (score < 0) calcScore();
        return score;
    } }

    public string Title { get {
        return titles[Score];
    } }

    public string CardName(int cardNum) {
        return cards[cardNum - 1].Name;
    }

    public string Text { get {
        return CardName(1) + " " +
            CardName(2) + " " +
            CardName(3) + " " +
            CardName(4) + " " +
            CardName(5);
    } }

    public override string ToString() {
        return Text;
    }

    private void cardsFromString(string handText) {
        char[] delims = { ' ' };
        string[] cardStrings = handText.Split(delims);
        for (int i = 0; i < cardStrings.Length; i++)
            cards[i] = new Card(cardStrings[i]);
    }

    private void holdCards(string holdString) {
        for (int i = 0; i < 6; i++) {
            int cardNum = i + 1;
            if (holdString.IndexOf(cardNum.ToString()) >= 0)
                isHold[cardNum - 1] = true;
        }
    }

    private void draw() {
        // remember which cards player has seen...
        Card[] seen = new Card[10];
        for (int i = 0; i < 5; i++) {
            seen[i] = cards[i];
        }

        int numSeen = 5;
        Random r = new Random();
        for (int i = 0; i < 5; i++) {

```

```

        if (!isHold[i]) {
            while (true) {
                cards[i] = new Card(r);
                if (containsCard(cards[i], seen, numSeen)) continue;
                break;
            }
            seen[numSeen++] = cards[i];
        }
    }
}

private bool containsCard(Card c, Card[] cs, int count) {
    for (int i = 0; i < count; i++)
        if (c.Equals(cs[i]))
            return true;
    return false;
}

private void calcScore() {
    // are cards all of the same suit?
    bool isFlush = true;
    int s = cards[0].Suit;
    for (int i = 1; i < 5; i++) {
        if (s != cards[i].Suit) {
            isFlush = false;
            break;
        }
    }

    // sort card values...
    int[] sortedValues = new int[5];
    for (int i = 0; i < 5; i++)
        sortedValues[i] = cards[i].Number;
    Array.Sort(sortedValues);

    // do we have a straight?
    bool isStraight = true;
    for (int i = 0; i < 4; i++) {
        if (sortedValues[i] + 1 != sortedValues[i+1]) {
            isStraight = false;
            break;
        }
    }

    // is it a straight to the ace?
    bool isTopStraight = (isStraight && sortedValues[4] == 14);

    // maybe it is a straight from the ace (i.e. A, 2, 3, 4, 5)
    if (!isStraight)
        if (sortedValues[0] == 2 &&
            sortedValues[1] == 3 &&
            sortedValues[2] == 4 &&
            sortedValues[3] == 5 &&
            sortedValues[4] == 14) // ace on top

```

```

        isStraight = true;

// now calculate score...
// royal flush...
if (isTopStraight && isFlush) {
    score = 10;
    return;
}
// straight flush...
if (isStraight && isFlush) {
    score = 9;
    return;
}
// four of a kind...
if (sortedValues[0] == sortedValues[1] &&
    sortedValues[1] == sortedValues[2] &&
    sortedValues[2] == sortedValues[3]) {
    score = 8;
    return;
}
if (sortedValues[1] == sortedValues[2] &&
    sortedValues[2] == sortedValues[3] &&
    sortedValues[3] == sortedValues[4]) {
    score = 8;
    return;
}
// full house...
if (sortedValues[0] == sortedValues[1] &&
    sortedValues[1] == sortedValues[2] &&
    sortedValues[3] == sortedValues[4]) {
    score = 7;
    return;
}
if (sortedValues[0] == sortedValues[1] &&
    sortedValues[2] == sortedValues[3] &&
    sortedValues[3] == sortedValues[4]) {
    score = 7;
    return;
}
// flush...
if (isFlush) {
    score = 6;
    return;
}
// straight...
if (isStraight) {
    score = 5;
    return;
}

```

```

    }

    // three of a kind...
    if (sortedValues[0] == sortedValues[1] &&
        sortedValues[1] == sortedValues[2]) {
        score = 4;
        return;
    }
    if (sortedValues[1] == sortedValues[2] &&
        sortedValues[2] == sortedValues[3]) {
        score = 4;
        return;
    }
    if (sortedValues[2] == sortedValues[3] &&
        sortedValues[3] == sortedValues[4]) {
        score = 4;
        return;
    }

    // two pair...
    if (sortedValues[0] == sortedValues[1] &&
        sortedValues[2] == sortedValues[3]) {
        score = 3;
        return;
    }
    if (sortedValues[0] == sortedValues[1] &&
        sortedValues[3] == sortedValues[4]) {
        score = 3;
        return;
    }
    if (sortedValues[1] == sortedValues[2] &&
        sortedValues[3] == sortedValues[4]) {
        score = 3;
        return;
    }

    // jacks or better...
    if (sortedValues[0] > 10 &&
        sortedValues[0] == sortedValues[1]) {
        score = 2;
        return;
    }
    if (sortedValues[1] > 10 &&
        sortedValues[1] == sortedValues[2]) {
        score = 2;
        return;
    }
    if (sortedValues[2] > 10 &&
        sortedValues[2] == sortedValues[3]) {
        score = 2;
        return;
    }
    if (sortedValues[3] > 10 &&

```

```

        sortedValues[3] == sortedValues[4]) {
            score = 2;
            return;
        }
        score = 0;
        return;
    }

    private Card[] cards = new Card[5];
    private bool[] isHold = {false, false, false, false, false};

    private static string[] titles = {
        "No Score",
        "",
        "Jacks or Better",
        "Two Pair",
        "Three of a Kind",
        "Straight",
        "Flush",
        "Full House",
        "Four of a Kind",
        "Straight Flush",
        "Royal Flush",
    };

    private int score = -1;
}
}

```

## 3.4 SIMPOK: A SIMPLE POKER GAME

Now it is time to build our first version of video poker. This version will provide a simple poker machine class which can deal and draw cards, but which ignores game histories and omits profit calculations for now. We also implement a simple console interface to this machine.

### 3.4.1 The Poker.SimpleMachine class

Listing 3.3 presents a class called `SimpleMachine` which represents a simple poker machine with the ability to deal and draw hands.

**Listing 3.3 The Poker.SimpleMachine class**

```

namespace Poker {
    public class SimpleMachine {
        public Hand Deal() {
            return new Hand();
        }
        public Hand Draw(Hand oldHand, string holdCards) {
            return new Hand(oldHand, holdCards);
        }
    }
}

```

```

        public Hand Draw(string oldHand, string holdCards) {
            return new Hand(oldHand, holdCards);
        }
    }
}

```

SimpleMachine is really just a wrapper class for constructing Hand objects. We'll build a more powerful machine with database support, and payout control, in the following chapter.

Appendix B contains the code for all the classes, which make up the poker engine, together with a makefile to build the DLL. These files can also be downloaded from <http://www.manning.com/grimes>. If, however, you wish to build the DLL with the classes presented so far, you can issue the following compiler command:

```
csc /t:library /out:poker.dll card.cs hand.cs simplemachine.cs
```

### 3.4.2 The SimPok console interface

Let's create a short console program to deal and draw cards. Listing 3.4 illustrates.

#### Listing 3.4 The SimPok program

```

// file      : simpok.cs
// compile   : csc /r:poker.dll simpok.cs

using System;
using Poker;

class SimPok {

    public static void Main() {
        new SimPok(); // start game
    }

    public SimPok() {
        Console.WriteLine("A simple poker game...");
        Console.WriteLine("Hit Ctrl-c at any time to abort.\n");
        machine = new SimpleMachine(); // create poker machine
        while (true) nextGame(); // play
    }

    private void nextGame() {

        Hand dealHand = machine.Deal(); // deal hand
        Console.WriteLine("{0}", dealHand.Text); // display it

        // invite player to hold cards...
        Console.Write("Enter card numbers (1 to 5) to hold: ");
        string holdCards = Console.ReadLine();

        // draw replacement cards...
        Hand drawHand = machine.Draw(dealHand, holdCards);
        Console.WriteLine(drawHand.Text);
        Console.WriteLine(drawHand.Title);
    }
}

```

```

        Console.WriteLine("Score = {0}\n", drawHand.Score);
    }

    private SimpleMachine machine;
}

```

The program starts by greeting the user and creating a new instance of `SimpleMachine`. Then it repeatedly calls `nextGame` until the user presses CTRL+C to abort. The `nextGame` method deals a hand, displays it to the user, and asks the user which cards to hold. (Cards are identified by their positions, 1 to 5.) The user's reply is captured in `holdCards` and cards are drawn by constructing a new hand from the old, as follows:

```

    Hand drawHand = machine.Draw(dealHand, holdCards);

```

Compile and execute this program, as shown in figure 3.4.

```

C:\DotNet\poker\SimPok>csc /r:poker.dll simpok.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR vers
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.

C:\DotNet\poker\SimPok>simpok
A simple poker game...
Hit Ctrl-c at any time to abort.

JS KH KS 6H 2H
Enter card numbers <1 to 5> to hold: 23
2S KH KS AH 3D
Jacks or Better
Score = 2

5S AS TC QD AD
Enter card numbers <1 to 5> to hold: _

```

**Figure 3.4**  
Compiling and running SimPok

## 3.5 COMPOK: A COM-BASED POKER GAME

Before .NET came along, we might have developed and deployed our poker machine as a COM object. Doing so would have enabled us to create various clients which use COM automation to play video poker. Since COM and .NET will likely coexist for some time to come, .NET provides the ability for both to interoperate. For example, the assembly registration utility, *regasm.exe*, allows us to register a .NET assembly as a COM object. Let's explore this as a deployment option with a simple COM-based version of video poker.

### 3.5.1 Registering the poker assembly as a COM object

Copy *poker.dll* to the *C:\WINNT\system32* directory and then execute *regasm.exe* to register it in the registry, as shown in figure 3.5.

```

C:\DotNet\poker\SimPok>copy poker.dll c:\winnt\system32
1 file(s) copied.

C:\DotNet\poker\SimPok>regasm c:\winnt\system32\poker.dll
RegAsm - .NET Assembly Registration Utility Version 1.0.2914.16
Copyright (C) Microsoft Corp. 2001. All rights reserved.

Types registered successfully

C:\DotNet\poker\SimPok>_

```

**Figure 3.5 Registering the Poker.dll assembly**

The *regasm.exe* utility reads the assembly metadata and makes the necessary entries in the registry to enable COM clients to create instances of the .NET types.

### 3.5.2 Console poker using COM and VBScript

Listing 3.5 presents a VBScript client which plays a COM-based version of our simple poker game.

#### Listing 3.5 The ComPok program

```

' file:          compok.vbs
' description:   VBScript poker game
' execute:      cscript compok.vbs

wscript.stdout.WriteLine "A simple poker game..."
wscript.stdout.WriteLine "Hit Ctrl-c at any time to abort"
wscript.stdout.WriteLine

set machine = wscript.createObject("Poker.SimpleMachine")
do while true ' play forever

    set dealHand = machine.Deal()
    wscript.stdout.WriteLine dealHand.Text

    wscript.stdout.write "Enter card numbers (1 to 5) to hold: "
    holdCards = wscript.stdin.ReadLine

    set drawHand = machine.Draw(dealHand, holdCards)
    wscript.stdout.WriteLine drawHand.Text
    wscript.stdout.WriteLine drawHand.Title
    wscript.stdout.WriteLine "Score = " & drawHand.Score
    wscript.stdout.WriteLine

loop

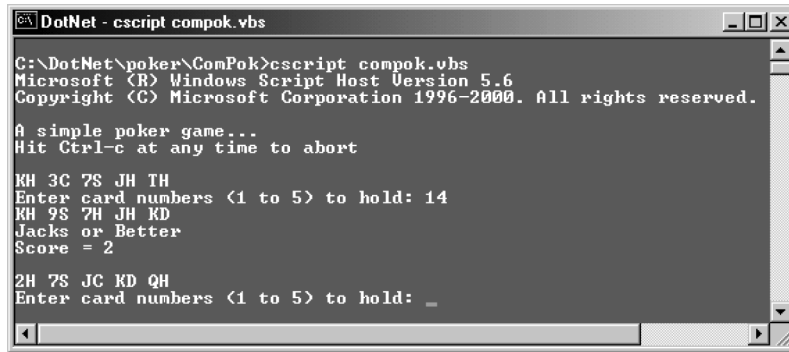
```

For our purposes, the most important line in the program is where we create a COM-based instance of the SimpleMachine:

```
set machine = wscript.createObject("Poker.SimpleMachine")
```

The fully qualified .NET type name, `Poker.SimpleMachine`, is used as the `ProgID` to identify the COM class in the registry.

We use the console version of the Windows Scripting Host to load and run this version of the poker game, as shown in figure 3.6.



```
DotNet - cscript compok.vbs
C:\DotNet\poker\ComPok>cscript compok.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

A simple poker game...
Hit Ctrl-c at any time to abort

KH 3C 7S JH TH
Enter card numbers <1 to 5> to hold: 14
KH 9S 7H JH KD
Jacks or Better
Score = 2

2H 7S JC KD QH
Enter card numbers <1 to 5> to hold: _
```

Figure 3.6 Running ComPok

### 3.5.3 RegAsm and the registry

When we run `regasm.exe`, all public types from the assembly are registered in the registry. To check this, run `regasm.exe` again, using the `/regfile:poker.reg` option to generate a registration file instead of updating the registry directly. (The `poker.reg` file can be used to install the component on another machine.) The file should contain an entry for `Poker.SimpleMachine` which looks something like:

```
[HKEY_CLASSES_ROOT
  \CLSID
    \{5F9EF3C3-6A12-3636-A11E-C450A65F3C0C}
      \InprocServer32]
@="C:\WINNT\System32\mscoree.dll"
"ThreadingModel"="Both"
"Class"="Poker.SimpleMachine"
"Assembly"="poker, Version=1.0.0.0,
           Culture=neutral,
           PublicKeyToken=null"
"RuntimeVersion"="v1.0.2904"
```

There should be a similar entry for `Poker.Hand`, but not for `Poker.Card` because the latter is internal to the assembly.

To unregister `poker.dll` enter:

```
regasm /unregister poker.dll
```

For readers who are familiar with COM+ services, .NET also provides a utility called `RegSvcs` which allows you to install a .NET assembly into a COM+ application. You'll find more information about this in the .NET and COM+ reference documentation.

## 3.6 IEPOK: AN INTERNET EXPLORER POKER GAME

Internet Explorer can download assemblies on demand and install them in the assembly download cache. This gives us the ability to install the poker engine directly from a Web page and to script a browser-hosted version of the game. So, before we leave our simple poker machine, let's create a more user-friendly interface by hosting the game inside Internet Explorer and providing a graphical interface similar to a real poker machine.

### 3.6.1 Downloading assemblies using Internet Explorer

We can deploy *poker.dll* on the Web server and can use the following `<object>` tag to install it directly from a Web page:

```
<object id=machine
      classid=http:poker.dll#Poker.SimpleMachine>
</object>
```

This causes Internet Explorer to download *poker.dll*, install it in the download cache, and instantiate a `SimpleMachine` object. In this example, the *poker.dll* assembly must be in the same virtual directory as the Web page. It is downloaded and activated without prompting the user, and without making any entries in the registry.

Let's explore this as a deployment option. First, we need to use Internet Services Manager to create a new virtual directory on the server. I called this directory *iepok* and mapped it to the local path *C:\DotNet\poker\IEPok*. Figure 3.7 shows the properties of this virtual directory.

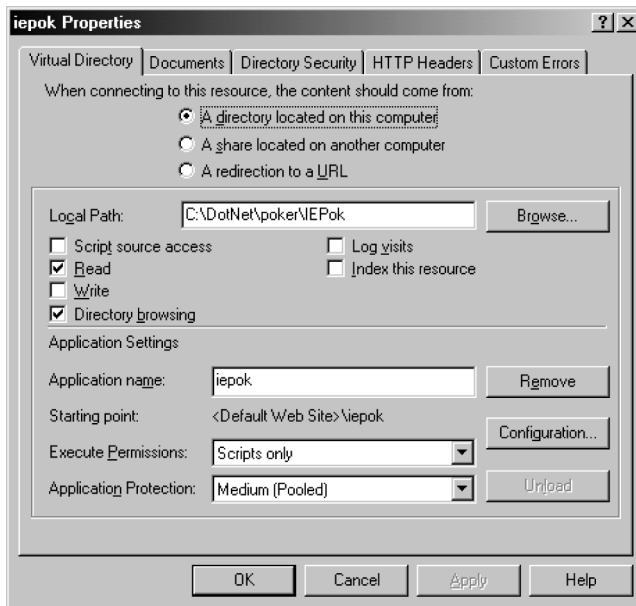


Figure 3.7  
The IEPOK virtual directory

### 3.6.2 Coding the IEPok application

Next, copy *poker.dll* to this new Web directory. We'll implement the Internet Explorer version of the game as the HTML file, *IEPok.html*, shown in listing 3.6. You'll find the GIF images of the playing cards in the download.

#### Listing 3.6 The IEPok.html file

```
<!-- file: IEPok.html -->
<html><head>

<object id=machine
        classid=http:poker.dll#Poker.SimpleMachine>
</object>
<script>
Hand = null;

function Cards() {
  if (Btn.value == "Deal") {
    Hand = machine.Deal();
    Hold1.checked = Hold2.checked = Hold3.checked =
      Hold4.checked = Hold5.checked = false;
    Btn.value = "Draw";
    Title.innerText = "Hold and Draw";
  } else { // draw cards...
    holdCards = "";
    if (Hold1.checked) holdCards += "1";
    if (Hold2.checked) holdCards += "2";
    if (Hold3.checked) holdCards += "3";
    if (Hold4.checked) holdCards += "4";
    if (Hold5.checked) holdCards += "5";
    Hand = machine.Draw(Hand, holdCards);
    Title.innerText = Hand.Title + " (" + Hand.Score + ")";
    Btn.value = "Deal";
  }
  Card1.src="images/" + Hand.CardName(1) + ".gif";
  Card2.src="images/" + Hand.CardName(2) + ".gif";
  Card3.src="images/" + Hand.CardName(3) + ".gif";
  Card4.src="images/" + Hand.CardName(4) + ".gif";
  Card5.src="images/" + Hand.CardName(5) + ".gif";
}
</script></head>

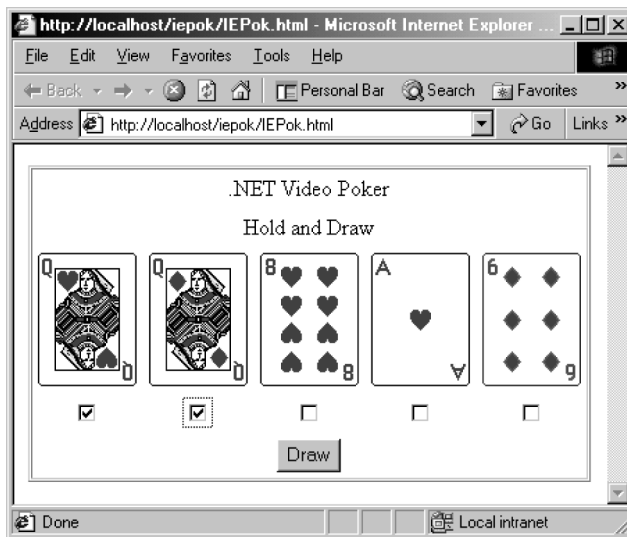
<body>
<table rules="none" border=1 cellpadding="4" cellspacing="1">
  <tr><td align=center colspan=5>.NET Video Poker</td></tr>
  <tr><td id="Title" align=center colspan=5>Click Deal</td></tr>
  <tr>
    <td></td>
    <td></td>
    <td></td>
    <td></td>
    <td></td>
  </tr>
</table>
</body>
</html>
```

```

</tr>
<tr>
  <td align=center><input type="checkbox" id="Hold1" /></td>
  <td align=center><input type="checkbox" id="Hold2" /></td>
  <td align=center><input type="checkbox" id="Hold3" /></td>
  <td align=center><input type="checkbox" id="Hold4" /></td>
  <td align=center><input type="checkbox" id="Hold5" /></td>
</tr>
<tr><td align=middle colSpan=5>
  <input type="button" value="Deal" id="Btn" onClick="Cards()"/>
</td></tr>
</table>
</body></html>

```

The `<object>` tag installs *poker.dll*. (Refer to chapter 2 for details on listing the contents of the download cache.) The remaining code is just standard HTML and JavaScript to create a table that displays 5 cards with checkboxes underneath. We use the same button for both dealing and drawing cards. All the user interface logic for the game is contained in the `Cards` JavaScript function. When the user clicks Deal/Draw, the program checks the button caption to see if it should deal or draw. If dealing, it simply deals a hand, clears the hold checkboxes, sets the button caption to Draw, and tells the user to hold and draw cards. If drawing, it examines the checkboxes to see which cards to hold, draws replacement cards, sets the button caption to Deal again, and tells the user the score. Refer to figure 3.8 to see how the game looks in the browser. The card images, which are available in the download for this book, are placed in the `images` subdirectory on the server. The image files follow the familiar two-character naming convention. For example, *qh.gif* is an image of the queen of hearts. Note that *cb.gif* is an image of the back of the cards.



**Figure 3.8**  
Internet Explorer-hosted  
video poker

### 3.7 DESIGNING A COMPLETE GAME

In the next chapter, we'll expand our poker machine to record game histories and enforce payout control. To do so, we'll use SQL Server to store the data. It has become common to design so-called N-tier client/server systems which partition the application into separate layers. A typical design often involves three tiers, or layers: data, logic, and interface. Since we intend to develop multiple, distributed interfaces for our poker machine, this kind of partitioning is an absolute requirement. We don't want to restrict application access to just Windows or Web users when .NET provides the building blocks for wider deployment. With a little extra work, we can support users coming from Windows, the Web, UNIX via telnet, a mobile phone, a PDA, and, using either remoting or XML Web services, we can expose the poker engine to other developers who wish to build their own customized application front ends.

#### 3.7.1 Video poker: the poker engine and its interfaces

Figure 3.9 shows an overview of the complete video poker application.

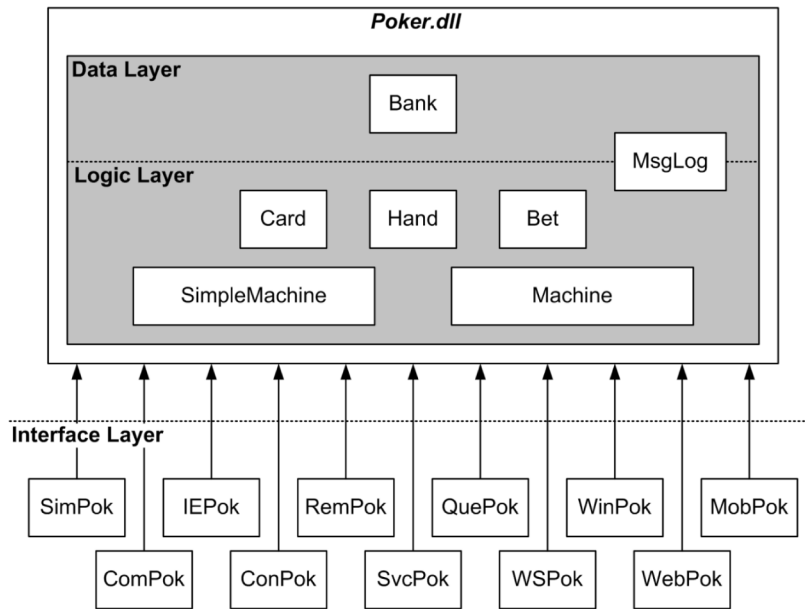


Figure 3.9 A model of the video poker application

The `poker.dll` assembly is logically divided into data and logic layers. We'll add the data layer, containing the `Bank` class, in the next chapter. The `MsgLog` class is just a utility class for logging errors and warnings in the Windows event log.

We've already developed the `Card`, `Hand`, and `SimpleMachine` classes. The full machine, a 3-tier application that supports betting and payout control, will be implemented in the `Machine` class, also in the next chapter.

Eleven versions of the poker game are shown:

- `SimPoker`—The simple console-based poker game already seen in this chapter.
- `ComPoker`—The VBScript/COM-based poker game already seen in this chapter.
- `IEPoker`—The Internet Explorer-based poker game already seen in this chapter.
- `ConPoker`—A 3-tier console poker game using SQL Server and ADO.NET.
- `RemPoker`—A client/server poker game which uses .NET remoting services.
- `SvcPoker`—A Windows service-based poker game.
- `QueuePoker`—A message queue-based poker game.
- `WSPoker`—A client/server poker game which uses XML Web services.
- `WinPoker`—A 3-tier Windows GUI poker game using Windows Forms, SQL Server, and ADO.NET.
- `WebPoker`—A Web server-based, ASP.NET poker game.
- `MobPoker`—A mobile poker game playable on a Web-enabled phone or PDA. We'll use .NET's Mobile Internet Toolkit to build this game.

In developing these game versions, we'll get a fairly complete tour of the different application models that .NET supports. We'll use the *poker.dll* assembly as a common poker engine behind each application.

### **3.8 SUMMARY**

In this chapter, we introduced our case study and developed a simple, console-based poker game. We also saw how to expose the poker machine as a COM object, and how to download and install it inside Internet Explorer. We also laid out a model for a complete implementation of the poker machine which leverages the features of the .NET platform to gain maximum deployment.

In the next chapter, we'll explore ADO.NET and build the data layer for the poker machine. We'll also put the finishing touches to the poker engine assembly.