Chapter 16

Case study – A Card Game

In the third case study 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 my introductory OOP book. This was translated into Java in the second edition [Budd 97], and revised once again in Java in a later book [Budd 98b]. The program presented here is yet another revision, this time translating back into C++ and making use of the Microsoft Foundation Classes for the graphical interface.

16.1 The Class Card

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 16.1.

Preprocessor directives (see Section 12.2) are placed at the beginning and the end of the file containing the class definition. These directives ensure that if the file is included more than once, the compiler will nevertheless see only one definition for the class Card. This is accomplished by testing to see if a name, one that hopefully occurs only in this context, has been defined. If not, then the name is given a definition (here a null definition), and the text is included. On subsequent inclusions the name will already be known, and the text between the ifndef and the corresponding endif statement will be ignored.

Within the class definition for Card two enumerated data types are defined for the discrete classes of suits and colors. The enumerated type declaration not only creates names for these values, but defines the range of elements they can hold.

The definition of the playing card abstraction is divided between an interface file, that defines the structure of the class, and an implementation file. In this particular case all but

```
# ifndef CARDH // ensure only included once
# define CARDH

//
//    Playing Card Abstraction
//

class Card {
public:
    enum Suits {Heart, Spade, Diamond, Club};
    enum Colors {Red, Black};

        // constructors
    Card (Suits sv, int rv) : s(sv), r(rv) { fup = false; }

        // return attributes
    int rank () { return r; }
    Suits suit() { return s; }
    bool faceup () { return fup; }
    Colors color ();

        // change attributes
    void flip () { fup = ! fup; }

private:
    const Suits s;
    const int r;
    bool fup;
};

# endif
```

Figure 16.1: The Definition of the class Card

one method is defined in the class body in the interface file. The one remaining method is found in the file Card.cpp, as follows:

```
# include "card.h"

Card::Colors Card::color ()
{
    if (suit() == Heart || suit() == Diamond)
        return Red;
    return Black;
}
```

The implementation file must include the interface file. Also a qualified name must be used to indicate the return type, as the return type is generally processed before the method name is read, and at that point the compiler does not yet know that the method is to be considered part of class Card.

Note that the class Card 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.

16.2 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 card 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 methods are declared as *pure virtual* methods (see Section 6.2). These will later be subclassed and replaced with functions that, in the case of the present application, will use the MFC (Microsoft Foundation Classes) to generate the graphical interface:

```
# ifndef CardViewH
# define CardViewH

# include "card.h"

//
//      CardView - Display a graphical representation of a card
//
```

```
class CardView {
public:
    static const int Width = 40;
    static const int Height = 70;

    virtual void display (Card * aCard, int, int) = 0;
    virtual void halfDisplay (Card * aCard, int, int) = 0;
};

# endif
```

The CardView class encapsulates a pair of static constants that represent the height and width of a card on the display. The Card class itself knows nothing about how it is displayed. Two static methods are prototyped. The first will display the entire face of a card at a given position on the display. The second will display only the upper half of a card.

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

16.3 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 that described in [Morehead 49]; variations on the basic game are numerous.

The layout of the game is shown in Figure 16.2. 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 face up; all other cards are face down.

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 face down, and are drawn one by one from the deck and placed, face up, 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 next higher

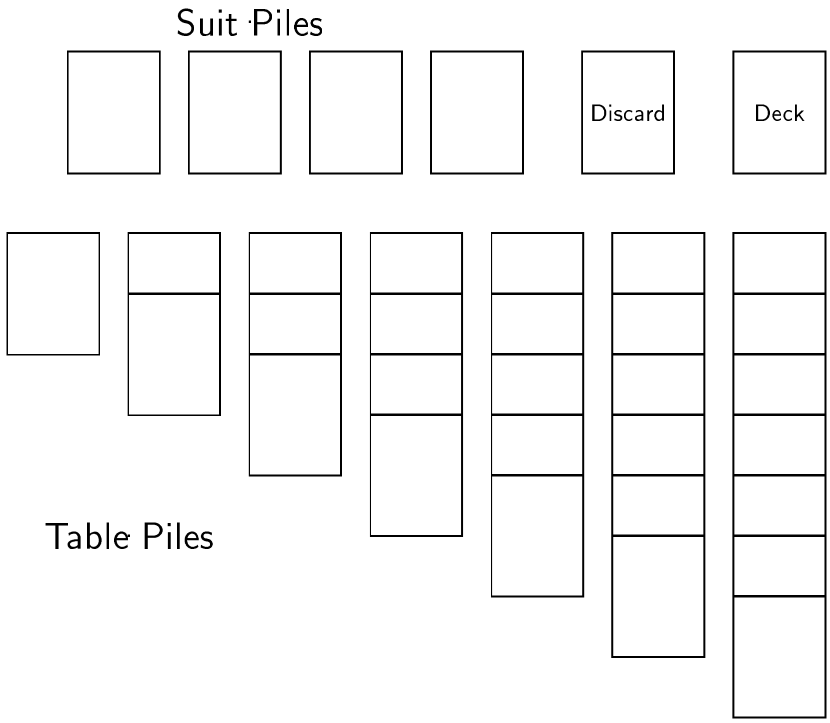

Figure 16.2: – Layout for the solitaire game.

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 face-up 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 face up. If a card is moved from a tableau, leaving a face-down card on the top, the latter card can be turned face up.

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.

16.4 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 linked list of the cards in the pile, and the operations of inserting and deleting elements from this linked list 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 16.3.

Each card pile maintains the coordinate location for the upper left corner of the pile, as well as a linked list containing the cards in the pile. The list abstraction from the Standard Template Library (see Chapter 9) is used for the latter. All these values are set by the constructor for the class, the first two explicitly by means of initializer classes and the last implicitly, since no other initialization instruction is provided to the compiler. The data fields are declared as protected and thus accessible to member functions associated with this class and to member functions associated with subclasses.

The class PopError is a nested class (see Section 4.7). The class declares no data fields, and no methods. The sole purpose of the class is to provide a unique identifier for the exceptions thrown by an attempt to read from an empty list.

The three functions top(), pop(), and empty() manipulate the list of cards, using functions provided by the list class. Two are provided with in-line definitions, while the top method is found in the implementation file:

```
Card * CardPile::top() throw (CardPile::PopError)
{
    if (cards.empty())
        throw new PopError();
    return cards.front();
}
```

As would be expected given the C++ philosophy of valuing efficiency over run-time checks, the list abstraction in the standard library is not obligated to report an error if an attempt is made to access the first element of an empty container. Thus, if we desire to make the program more robust we must explicitly test for this condition ourselves. Doing so makes the program more reliable, at the expense of slowing down executions of the top and pop operations.

To catch the potential stack underflow, the main program will at some point be surrounded by a try clause:

```
    try {
```

```
# ifndef CardPileH
# define CardPileH

# include "card.h"
# include <list>

//
//     CardPile – representation of a Pile of Cards
//

class CardPile {
public:
    CardPile (int xl, int yl) : x(xl), y(yl) { }

        // exception on pop from empty stack
    class PopError { };

        // drawing cards from pile
    bool empty () { return cards.empty(); }
    Card * top () throw (PopError);
    Card * pop () throw (PopError)
        { Card * result = top(); cards.pop_front(); return result; }

        // virtual methods that can be overridden
    virtual bool includes (int, int);
    virtual void addCard (Card *);
    virtual void display (CardView &);
    virtual bool canTake (Card *);
    virtual void select ();

protected:
    list<Card *> cards;
    const int x; // location of display
    const int y;
};

... // definitions of subclasses

# endif
```

Figure 16.3: – Description of the class CardPile.

```
    ... // execute the program
} catch (CardPile::PopError & e) {
    ... // handle the exception
    }
```

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(Card) asks whether it is legal to place a card on the given pile. A card can be added to a foundation pile, for instance, only if it is an ace and the foundation is empty, or if the card is of 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 face up.

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 face-up 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 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 five operations and five 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	×		×	×	×

16.4.1 The Default Card Pile

We will examine each of the subclasses of CardPile in detail, pointing out various uses of object-oriented features and differences between the C++ definitions and similar Java procedures as they are encountered. 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:

```
bool CardPile::includes (int tx, int ty)
{
        // default behavior : is point within bounds of card?
    return x <= tx && tx <= x + CardView::Width &&
        y <= ty && ty <= y + CardView::Height;

}

void CardPile::addCard (Card * aCard)
{
        // default behavior: push card on front of stack
    cards.push_front(aCard);

}

void CardPile::display (CardView & cv)
{
        // default behavior: print topmost card
    if (empty())
        cv.display(0, x, y);
    else
        cv.display(top(), x, y);
}

bool CardPile::canTake (Card *)
{
        // default behavior: just say no
```

```
    return false;
}

void CardPile::select ()
{
        // default behavior: do nothing
}
```

16.4.2 The Suit Piles

The simplest subclass is the class SuitPile, which represents the pile of cards at the top of the playing surface, the pile being built up in suit from ace to king. The interface for this class is as follows:[1]

```
class SuitPile : public CardPile {
public:
    SuitPile (int xl, int yl) : CardPile(xl, yl) { }
    bool canTake (Card *);
};
```

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. Because this action is simple, the method is defined in-line as part of the class description. The method canTake determines whether or not 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). The code that provides this behavior is more complex, and is therefore defined in the implementation file, rather than in the class definition itself:

```
bool SuitPile::canTake (Card * aCard)
{
        // can take ace if empty
    if (empty())
        return aCard->rank() == 1;
        // otherwise must be next card in suit
    Card * topCard = top();
    return (aCard->suit() == topCard->suit()) &&
```

[1] In the code distributed with this case study the interfaces for all the various types of card piles have been combined into a single file, so as to reduce the number of files needed to create the program.

```
        (aCard->rank() == 1 + topCard->rank());
}
```

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 linked list. To display the pile only the topmost card is drawn.

16.4.3 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 using an array of card values passed as argument to the constructor. 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.

```
class DeckPile : public CardPile {
public:
    DeckPile (int, int, Card * []);
    void select ();
};
```

The implementation of the constructor shows how initialization clauses are written when code is not being defined in-line:

```
DeckPile::DeckPile (int xl, int yl, Card * orig[ ])
    : CardPile(xl, yl)
{
    for (int i = 0; i < 52; i++) {
        Card * theCard = orig[i];
        if (theCard->faceup())
            theCard->flip();
        addCard(theCard);
    }
}
```

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, turing it face up 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

```
# include "CardPile.h"

//
//      Game – Game Manager
//

class Game {
public:
        // constructor, initialization and Destructor
    void init ();
    ~Game ();

        // access to the piles
    CardPile * deckPile () { return piles[0]; }
    CardPile * discardPile () { return piles[1]; }
    CardPile * suitPile (int i) { return piles[2+i]; }
    CardPile * tableau (int i) { return piles[6+i]; }

        // handling actions
    void repaint(CardView &);
    void mouseDown (int, int);

private:
    CardPile * piles[13];
};
```

Figure 16.4: Definition of Application Class

earlier version of the game [Budd 97]. In Java the equivalent of global variables is a series of static values, as in the program described in [Budd 98b].

In this version of the program we will reduce the number of global variables to one, which will be a global that represents a "game manager". The class definition for the game manager is shown in Figure 16.4. A discussion of some of the details of this class will be postponed until after the description of the various card piles.

The global variable gameManager will be defined as an instance of this class. This variable is declared as external to the class deck implementation file.

```
# include "Game.h" // include description of game manager
extern Game gameManager;

void DeckPile::select ()
```

```
{
        // move topmost card to discard stack
    if (empty())
        return;
    gameManager.discardPile()->addCard (top());
    pop();
}
```

An important point must be made here concerning circularities. It is not legal for interface files to have a circular structure. Doing so would put the preprocessor into an infinite loop, since preprocessing is performed before the analysis of the source program. It would at first appear that circularity is inevitable here, since the application class (SolitareApplication) must of necessity know about the various types of card piles, and the card piles must know about the application class.

What saves us in this case is the separation between interface and implementation. It is only the *implementation* of class DeckPile that must know about the game application, the *interface* says nothing about it. Thus, the interface file for CardDeck need not include the interface file for the application, however this file is needed by the implementation file for the abstraction.

As circularities of this sort are common in programs, the reader would be well advised to study this situation until they are clear where the various different files, both interface and implementation, are being used.

16.4.4 The Discard Pile

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

```
class DiscardPile : public CardPile {
public:
    DiscardPile (int xl, int yl) : CardPile(xl, yl) { }
    void addCard (Card *);
    void select ();
};
```

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.

```
void DiscardPile::select ()
```

```
{
    if (empty())
        return;
    Card * topCard = top();
    for (int i = 0; i < 4; i++)
        if (gameManager.suitPile(i)->canTake(topCard)) {
            gameManager.suitPile(i)->addCard(topCard);
            pop();
            return;
            }
    for (int i = 0; i < 7; i++)
        if (gameManager.tableau(i)->canTake(topCard)) {
            gameManager.tableau(i)->addCard(topCard);
            pop();
            return;
            }
}
```

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 face up. After satisfying this condition, the code in the parent class is invoked to add the card to the pile. A fully qualified name is necessary to avoid the confusion with the addCard method being defined:

```
void DiscardPile::addCard (Card * aCard)
{
    if (! aCard->faceup())
        aCard->flip();
    CardPile::addCard(aCard);
}
```

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.

16.4.5 The Tableau Piles

The most complex of the subclasses of CardPile is that used to hold a tableau, or table pile. The interface for this class redefines nearly all of the virtual methods defined in ClassPile:

```
class TablePile : public CardPile {
public:
    TablePile (int xl, int yl, int p);

    bool canTake (Card * aCard);
    bool includes (int tx, int ty);
    void display (CardView & cv);
    void select ();
};
```

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 face up.

```
TablePile::TablePile(int xv, int yv, int p) : CardPile(xv, yv)
{
        // copy right number of cards into deck
    for (int i = 0; i < p; i++) {
        Card * aCard = gameManager.deckPile()->top();
        if (aCard->faceup())
            aCard->flip();
        addCard(aCard);
        gameManager.deckPile()->pop();
    }
        // flip topmost card
    top()->flip();
}
```

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:

```
bool TablePile::canTake (Card * aCard)
{
    if (empty())
        return aCard->rank() == 12;
```

```
    Card * topCard = top();
    return (aCard->color() != topCard->color()) &&
        (aCard->rank() == topCard->rank() - 1);
}
```

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:

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

When the pile is selected, the topmost card is flipped if it is face down. If it is face up, 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:

```
void TablePile::select ()
{
        // if empty, do nothing
    if (empty()) return;

        // if face down, then flip
    Card * topCard = top();
    if (! topCard->faceup()) {
        topCard->flip();
        return;
    }

        // else see if any pile can take card
    for (int i = 0; i < 4; i++)
        if (gameManager.suitPile(i)->canTake(topCard)) {
            gameManager.suitPile(i)->addCard(pop());
            return;
        }
    for (int i = 0; i < 7; i++)
        if (gameManager.tableau(i)->canTake(topCard)) {
            gameManager.tableau(i)->addCard(pop());
            return;
        }
}
```

Finally, to display a pile all the underlying cards are displayed in half form, and the topmost card is displayed full:

```
void TablePile::display (CardView & cv)
{
    if (empty())
        CardPile::display(cv);
    else {
        int lx = x;
        int ly = y;
        list<Card *>::iterator cptr = cards.end();
        list<Card *>::iterator front = cards.begin();
        for (--cptr; cptr != front; --cptr) {
            cv.halfDisplay(*cptr, lx, ly);
            ly += CardView::Height/2;
            }
        cv.display(*front, lx, ly);
    }
}
```

16.5 Playing the Polymorphic Game

The game manager class was described already in Figure 16.4. The game manager stores the various card piles in an array, one that is declared as pointers to CardPile, although in fact the values are polymorphic and hold a variety of different types of card piles. These values are initialized in the method init, as follows:

```
void Game::init ()
{
        // first, create a deck and randomize it
    int j = 0;
    for (int i = 1; i <= 13; i++) {
        originalDeck[j++] = new Card(Card::Diamond, i);
        originalDeck[j++] = new Card(Card::Spade, i);
        originalDeck[j++] = new Card(Card::Heart, i);
        originalDeck[j++] = new Card(Card::Club, i);
        }

    randomInteger swapper; // declare the random function object
    random_shuffle(originalDeck, originalDeck+52, swapper);
```

```
        // then initialize each of the deck piles
    piles[0] = new DeckPile(335, 5, originalDeck);
    piles[1] = new DiscardPile(268, 5);
    j = 2;
    for (int i = 0; i < 4; i++) {
        piles[j++] = new SuitPile(15 + 60 * i, 5);
        }
    for (int i = 0; i < 7; i++) {
        piles[j++] = new TablePile(5 + 55 * i, 80, i+1);
        }
}
```

There is a subtle reason why initialization must be performed by a method, and not by the constructor for this class. The reason is that the constructor for TablePile, described in previous section, uses the global variable gameManager, which is an instance of this class. The C++ language does not consider an object initialized until the constructor has completely finished. Thus, the value of gameManager used by the TablePile constructor may not be available at the point it is needed. However, a careful reading of the init method shows that the deck pile is the first pile initialized, and therefore if executed as part of a function will be available for use by the initialization routine for table pile.

The method random_shuffle, used to shuffle the deck, is one of the generic algorithms provided by the standard library. This particular function requires a function object that will generate random integer values from a given range. Such an object can be produced by the following class declaration:

```
class randomInteger {
public:
    unsigned int operator () (unsigned int max) {
        unsigned int rval = rand();
        return rval % max;
        }
};
```

Space for the card values is created in the method init. This space is freed by the destructor:

```
Game::~Game()
{
        // free up all the old card values
    for (int i = 0; i < 52; i++)
        delete originalDeck[i];
```

```
}
```

Memory management techniques, such as reference counting (see Chapter 5) are not necessary in this case since the cards are neither created nor destroyed while the application is in 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:

```
void Game::repaint(CardView & cv)
{
        // simply repaint each of the card decks
    for (int i = 0; i < 13; i++)
        piles[i]->display(cv);
}
```

The method display will be different, depending upon the actual type of card pile. Similarly, to repond to a mouse down, the manager simply cycles through the list of card piles:

```
void Game::mouseDown(int x, int y)
{
    for (int i = 0; i < 13; i++)
        if (piles[i]->includes(x, y)) {
            piles[i]->select();
            return;
        }
}
```

16.6 The Graphical User Interface

We have taken pains in the development of this program to isolate the details of the graphical interface. This is because of all the elements of a program, the user iterface is the most likely to require change as new graphical libraries are introduced or existing libraries are changed. In this application, all the graphical aware routines can be isolated in one file.

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 MFCardView:

```
class MFCardView : public CardView {
public:
```

```
        // constructor saves drawing context
    MFCardView (CPaintDC & idc) : dc(idc) { }

        // implement the interface
    void display (Card * aCard, int, int);
    void halfDisplay (Card * aCard, int, int);

private:
    CPaintDC & dc; // drawing context
    void paintCard(Card * aCard, int, int);
};
```

This is not a text on fancy 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:

```
void MFCardView::display (Card * aCard, int x, int y)
{
    dc.Rectangle (x, y, x + Width, y + Height);
    paintCard(aCard, x, y);
}

void MFCardView::halfDisplay (Card * aCard, int x, int y)
{
    dc.Rectangle (x, y, x + Width, y + Height/ 2);
    paintCard (aCard, x, y);
}

void MFCardView::paintCard (Card * aCard, int x, int y)
{
    char buffer[80];
    if (aCard == 0) {
        strcpy(buffer,"mt"); // mt = empty
        dc.SetTextColor(RGB(0,255, 0)); // green
        dc.TextOut(x+5, y+5, buffer, strlen(buffer));
    }
    else {
        if (aCard->faceup()) {
            char suitCard;
            switch (aCard->suit()) {
                case Card::Heart: suitCard = 'H'; break;
                case Card::Spade: suitCard = 'S'; break;
```

```
                case Card::Diamond: suitCard = 'D'; break;
                case Card::Club: suitCard = 'C'; break;
            }
            switch (aCard->rank()) {
                case 1: sprintf(buffer,"A %c", suitCard); break;
                case 11: sprintf(buffer, "J %c", suitCard); break;
                case 12: sprintf(buffer, "Q %c", suitCard); break;
                case 13: sprintf(buffer, "K %c", suitCard); break;
                default: sprintf(buffer, "%d %c", aCard->rank(), suitCard);
                    break;
            }
            if (aCard->color() == Card::Red)
                dc.SetTextColor(RGB(255, 0, 0)); // red
            else
                dc.SetTextColor(RGB(0, 0, 0)); // black
            dc.TextOut(x+5, y+5, buffer, strlen(buffer));
        }
        else {
            strcpy(buffer, "back");
            dc.SetTextColor(RGB(255, 255, 0)); // yellow
            dc.TextOut(x+5, y+5, buffer, strlen(buffer));
        }
    }
}
```

Graphics in the MFC is based around a *display context*, here stored in in the variable dc. Details of the graphical output routines provided by the MFC 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 MFC framework are created by means of two classes, a window class and an application class. The two are linked, and the creation of the latter involves the initialization of the game manager. The main window is an instance of an MFC provided window, called CFrameWnd:

```
class SolitareMainWindow : public CFrameWnd  {
public:
    SolitareMainWindow() {  Create(NULL, "Solitare Game"); }
    afx_msg void OnPaint();
    afx_msg void OnLButtonDown (UINT flag, CPoint loc);
    DECLARE_MESSAGE_MAP()
};
```

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.

```
void SolitareMainWindow::OnPaint()
{
    CPaintDC dc(this);
    MFCardView cv(dc);

    gameManager.repaint(cv);
}

void SolitareMainWindow::OnLButtonDown(UINT flag, CPoint loc)
{
    try {
        gameManager.mouseDown(loc.x, loc.y);
    }
    catch (CardPile::PopError & e) { } // do nothing on error

    InvalidateRect(NULL);
}
```

The application class is again subclassed from a MFC provided parent, called CWinApp. The child class must redefine a virtual method named InitInstance. In this method the application class is linked to the window class, and the window class is told to display. Other application specific initialization, such as the initialization of our game manager, can be placed here. Creating an instance of the application class begins the application.

```
class SolitareApplication : public CWinApp  {
public:
    BOOL InitInstance();
};


SolitareApplication theApp;

BOOL SolitareApplication::InitInstance()
{
    gameManager.init();
    m_pMainWnd = new SolitareMainWindow();
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow();
}
```

We have in this simple application only scratched the surface of the functionality provided by the MFC system. However, the details of how MFC programs are created are complicated, and beyond the issues being discussed here. A good introduction to the MFC system is provided by Schildt [Schildt 96].

Test Your Understanding

1. Explain the purpose of the ifndef/endif pair that surrounds the interface file for class Card (Figure 16.1).

2. Why is the color method of class Card defined in the implementation file, while all other methods are defined in the interface file.

3. Why should data view classes be separated from data value classes?

4. Explain why the containers provided by the standard library, such as list or vector, do not detect an attempt to access an element in an empty container.

5. Explain why the Game class cannot be initialized in a constructor.