8.5 Object-Oriented Design

When you are faced with an open-ended problem of designing a program, object-oriented techniques can help. In many situations data elements can be organized into classes, which are usually easy to implement. If you can express the central algorithms of the problem in terms of these classes you probably have a good design for the program. For example, a program that records students grades (a virtual Registar program) might have a Person class, with subclasses for Students and Faculty, and a Course class, where a course would have an instructor and a list of Students. Each Student would have a current Schedule, with a list of classes, and a Transcript, which would include a list of class names and grades. A Student taking a Course would be represented by adding the student to the Course object's list of students,then at the end of the term getting a grade from the instructor and the course name and grade being added to the Student's transcript. By the time you implemented all of these classes the program would be mostly written.

For another example, suppose you are creating a predator-prey simulation. You would need a Predator class that describes the predator's ability to move around and locate the prey, and the predator's ability to kill nearby prey. You would have a corresponding Prey classes that describes the prey's ability to avoid being eaten. You might have an Environment class that both the Predator and Prey classes interact with. And you might have a World class that consists of a large grid, every cell of which has an Environment object and possibly one or more Predator and Prey objects.

Once you have identified possible classes for your program, try to write a brief description of each class. The closer the classes are to physical objects, or to objects that could be physical, the easier they will be to code. Try to write down the data elements of each class, and what kinds of Python representation each will have. Then write a description of each of the major methods of each class. You should do all of this before you start coding If you can describe the functionality of your program in terms of the class methods and you can see how to implement each of the methods, you have a winning program design that should be straightforward to implement. If you can't do these things there is no point in starting the coding. Try looking for a solution in terms of different class breakdowns.

As a demonstration of this we will develop a card game. This game will be played with a standard deck of 52 playing cards, with 13 cards in each of the suits Hearts, Spades, Diamonds and Clubs. To avoid implementing lengthy rules (which is better, a Full House or a Flush??) the game we will implement is the children's game War – on each round all of the players turn over their next card; the highest card wins. If there is a tie the players go again. The player who wins the most cards wins the game. This is not a very interesting game to play and it is even more uninteresting too watch, but the techniques and much of the code we will develop could be used to implement more interesting card games, from Go Fish to Texas Hold 'Em.

Our game will need 3 classes:

- class Card represents a single playing card. The data Card needs is the suit and name of the card. The names will be strings: "2", "3", "4", ..., up to "10", "Jack", "Queen", "King" and "Ace". For suits we could easily use the strings "Hearts", "Spades" and so forth, but for fun we will take advantage of Python's use of Unicode and use graphical representations of the suits. The Heart symbol is **chr**(9825), Spades is char(9824), Clubs is char(9827), and Diamonds is **chr**(9826). Cards don't do anything, so our only methods will be a constructor, a __str__ () method for printing, and a few comparison methods.

- class Deck will represent a deck of 52 cards. The only data is a list of the cards currently in the deck. Besides a constructor we need a method for shuffling the deck and another for dealing cards to players.

- class Player will represent one player of the game. This is a very simple class that keeps track of the player's name, current hand, and points. The only method this needs other than a constructor is a method that reveals the next card in the player's hand.

We will begin with the Card class. We'll define constants at the top that give the Unicode characters for our suits; this is the only place in the entire program we'll need to refer to these characters. Elsewhere we can refer to these suits as Card.HEARTS, Card.SPADES, and so forth. The constructor just assigns values; the real work has to be done by whoever calls the constructor.

Here is the complete code for the Card class:

```
class Card:
    HEARTS = chr(9825)
    SPADES = chr(9824)
    CLUBS = chr(9827)
    DIAMONDS = chr(9826)

    def __init__(self, name, suit, value):
        self.name = name
        self.suit = suit
        self.value = value

    def __str__(self):
        return "%s %s" %(self.name, self.suit)

    def __lt__(self, x):
        if self.value < x.value:
            return True
        else:
            return False

    def __gt__(self, x):
        return self.value > x.value

    def __eq__(self, x):
        return self.value == x.value
```

The Card class

Note the __lt__ () and __gt__ () methods. These give two different ways to write similar comparisons. In __lt__ () we do the comparison with <, look at the result, and decide what to return. In __gt__ () we realize that what we need to return is the result of the > comparison; we could just as easily have written __lt__ () as

```
    def __lt__(self, x):
        return self.value < x.value
```

The Deck class only has three methods, but each is more substantial than anything in the Card class. The constructor isn't difficult, but it requires some organization. We start by initializing the instance variable that will hold the list of cards in the deck to the empty list: self.cards=[]. Then, for each suit we add each of the number cards. Note that if x is a number betweem 2 and 10, str(x) is the string version of the number: "2" for 2, "10" for 10. After the number cards we add the Jack, Queen, King and Ace. The number on the number cards is the card's numeric value for comparison purposes; the value of a Jack is 11, that for a Queen is 12, for a King is 13 and for an Ace is 14. Note that in some card games an Ace is low, and in some it is either high or low; in ours it is always high. It would be possible to design a different class that had options for how the Ace is treated.

Here is the constructor code:

```
def __init__(self):
    self.cards = []
    for suit in [Card.HEARTS, Card.SPADES,
                 Card.DIAMONDS, Card.CLUBS]:
        for number in range(2, 11):
            self.cards.append(Card(str(number),
                                   suit, number))
        self.cards.append(Card("Jack", suit, 11))
        self.cards.append(Card("Queen", suit, 12))
        self.cards.append(Card("King", suit, 13))
        self.cards.append(Card("Ace", suit, 14))
```

In most card games the dealer cycles around all of the players giving each one card until everyone has the right number of cards. This guards against poorly shuffled decks and also provides some protection against a malicious dealer stacking the deck. We are not concerned about either of those problems since we will control how the cards are dealt, so it seems easier to have the Deal() method take the appropriate number of cards off the top of the deck and return them as a list of cards. We must remember to remove them from the deck so they can't be given to any other player. The only issue is what to do if the deck runs out of cards. Our solution is to give as many cards as the deck has, though you could make an argument for throwing an exception or taking other drastic action. Here is the resulting Deal method.

```
def __init__(self):
def Deal(self, n):
    # returns a list of n cards
    L = []
    for i in range( 0, n):
        if len(self.cards) > 0:
            card = self.cards[0]
            del self.cards[0]
            L.append(card)
    return L
```

The remaining method for the Deck class is Shuffle(). There are simple ways to randomize the elements of a list, but we are going to use a formal shuffling algorithm that has been proven to be fair: if the random number generator we use is good, our shuffle algorithm produces every possible ordering of the list and all of these orderings are equally likely. This is known as the *Fisher-Yates* *shuffle* since it was first defined by statisticians Ronald Fisher and Frank Yates in 1938. It is also called the *Knuth shuffle* since it was popularized by Donald Knuth in *The Art of Computer Programming.* The idea behind this algorithm is simple. We start at the end of the list and work our way towards the front. At each step we swap the current element with a random element closer to the front. When this is finished, the list is sorted:

```
def Shuffle(self):
    n = len(self.cards)-1
    while n > 0:
        k = randint(0, n)
        # swap the kth and nth cards
        A = self.cards[k]
        B = self.cards[n]
        self.cards[k] = B
        self.cards[n] = A
        n = n - 1
```

Altogether, here is the complete Deck class:

```
from Card  import *
from random import *

class Deck:
    def __init__(self):
        self.cards = []
        for suit in [Card.HEARTS, Card.SPADES,
                     Card.DIAMONDS, Card.CLUBS]:
            for number in range(2, 11):
                self.cards.append(Card(str(number),
                    suit, number))
            self.cards.append(Card("Jack", suit, 11))
            self.cards.append(Card("Queen", suit, 12))
            self.cards.append(Card("King", suit, 13))
            self.cards.append(Card("Ace", suit, 14))

    def Deal(self, n):
        # returns a list of n cards
        L = []
        for i in range( 0, n):
            if len(self.cards) > 0:
                card = self.cards[0]
                del self.cards[0]
                L.append(card)
        return L

    def Shuffle(self):
       n = len(self.cards)-1
       while n > 0:
           k = randint(0, n)
           # swap the kth and nth cards
           A = self.cards[k]
           B = self.cards[n]
           self.cards[k] = B
           self.cards[n] = A
           n = n - 1
```

The Deck class

With just this much we can implement a simple game I call *Cut the Deck*. We make a deck and two players each "cut the deck" by choosing an index between 0 and 51. The one who gets the higher card wins. If the cards have the same value they go again.

```
# This is a simple game simulating two people
# cutting a deck of cards.
# The game continues until one gets a higher
# card than the other.
from Deck import *

def cut(deck, player):
    index = eval( input("Where does %s cut the deck? "
                        % player))
    if index >= 52:
        index = 52
    elif index < 0:
        index = 0
    return deck.cards[index]

def main():
    D = Deck()
    player1 = input( "Who is the first player? " )
    player2 = input( "Who is the second player? " )
    done = False
    while not done:
        D.Shuffle()
        card1 = cut(D, player1)
        print( "%s draws %s" %(player1, card1) )
        card2 = cut(D, player2)
        print( "%s draws %s" %(player2, card2))

        if card1 > card2:
            print( "%s wins!" %player1 )
            done = True
        elif card2 > card1:
            print( "%s wins!" %player2 )
            done = True
        else:
            print( "Tie; play again." )

main()
```

Program 8.5.1: Game CutTheDeck

For our *War* game we need a player class. This will keep track of the player's name and hand of cards. In one round of the game each player plays one card;

since this involves multiple players it isn't a method of the Player class. Instead we give the class a method for playing one card, and put off the rest for our final implementation of the game. Here is the player class:

```
from Deck import *

class Player:
    def __init__(self, name, deck, numCards):
        self.name = name
        self.hand = deck.Deal(numCards)
        self.points = 0

    def nextCard(self):
        card = self.hand[0]
        del self.hand[0]
        print("%s plays %s" %(self.name, card))
        return card
```

The Player class

Finally, we need to implement the game itself. This has a play() function that has each player play one card. The cards are compared. If there is a winner the appropriate number of points (the number of cards won) are added to that player's total score; if there is no winner the play continues. Naturally, a loop describes this; it continues until one of the players wins or the players run out of cards.

The rest of the game is described in the main() function. This gets the players' names, constructs the players, and calls the play() method in a loop until the players run out of cards. At the end it reports the winner: the player with the highest score. Here is the full game:

```
from Player import *

def play(player1, player2):
    points = 0
    done = False
    while not done:
        c1 = player1.nextCard()
        c2 = player2.nextCard()
        points = points+2
        if c1 > c2:
            player1.points = player1.points + points
            print( "%s has %d points\n" %
                    (player1.name, player1.points))
            done = True
        elif c2 > c1 :
            player2.points = player2.points + points
            print( "%s has %d points\n" %
                    (player2.name, player2.points))
            done = True
        elif len(player1.hand) == 0:
            done = True

def main():
    deck = Deck()
    deck.Shuffle()
    player1 = Player( "bob", deck, 26)
    player2 = Player("marvin", deck, 26)

    while len( player1.hand) > 0:
        play( player1, player2)

    if player1.points == player2.points:
        print("Tie game")
    else:
        if player1.points > player2.points:
            winner = player1
        else:
            winner = player2
        print( "%s wins with %d points" %
                (winner.name, winner.points))

main()
```

Program 8.5.2: Game CutTheDeck

Notice how easy the game was to implement after all of our classes were defined. This is a sign of a successful object-oriented design. If you can't use the classes to solve the overarching problem you are working on, you probably haven't yet found a good class system for your problem.