Object-Oriented Design

The Game of War

Illustrate the process of designing an object-oriented solution to a small problem by simulating War, a two-person game for children that uses a deck of playing cards.

The rules of the game act as an initial specification of the problem.

> A card deck consists of 52 playing cards, each of which has a numeric value between 1 and 13, called the rank of the card, and one of four suits (clubs, diamonds, hearts, and spades).
>
> The values 1, 11, 12, and 13 have special names: ace, jack, queen, and king, respectively.
>
> To play the game we deal the entire deck of cards to two players.
>
> Each player has a pile of 26 cards face down on the table.
>
> During a turn of the game, the players show the top cards from their piles, and the player with the higher-ranking card wins both of the cards, putting them face-up in a pile.
>
> For the purpose of comparison, aces count as highest.
>
> If the cards shown by the players have the same rank, each player places the number of cards equal to that rank face down on the table.

The last cards played by each player are turned over and compared.

The higher value wins all of the cards that have been played this turn.

If another tie occurs, the process is repeated.

When a player finishes playing all of the cards in his or her playing pile, the cards won in previous turns are turned over and become a new playing pile.

If either player cannot complete a turn for lack of cards, the other player wins the game.

The goal of this program is to simulate the playing of a game of War with two players, named Ernie and Burt.

Analysis

After reviewing the specification of the problem, several points need clarification.

1. Implied in the game is the fact that we need to shuffle the deck of cards before the cards are dealt to the players.
2. Note that each player has two piles of cards: The pile of cards from which the player takes cards to play a turn and a pile that holds the cards won during the turns. When the playing pile becomes exhausted, the pile of cards won by the player so far becomes a new playing pile.

One way to identify the concepts that may become the objects of the program is to analyze the specification in terms of parts of speech.

- Nouns in the specification become objects in the solution, properties of objects, or values of properties.
- Verbs in the specification frequently describe the behavior (methods) of the objects.

Begin the analysis by identifying nouns in the specification.

card deck	game	last card
playing card	player	tie
numeric value	pile	process
rank of the card	turn	both of the cards
suit	top card	cards that have been played
clubs, diamonds	higher-ranking card	cards won in previous turns
hearts, spades	same rank	playing pile
ace, jack, queen, king	number of cards	lack of cards

Concepts

During a first pass several of the nouns seem to suggest concepts that have a good chance of being objects in the program.

card deck, playing card, game, player, and pile

The concept of a game may seem strange at first, but we need some controlling force to manage the turns of the game and evaluate the results.

We know that a card deck is made up of 52 playing cards, and that the game will have two players.

From the observation made earlier, we know that each player will have two piles of cards: a playing pile and a pile of cards won.

Note that many of the other nouns in the list are variations of these five basic ideas: “top card”, “higher-ranking card”, “last card”, and “both of the cards” all involve the concept of playing card.

The nouns “cards won in previous turns” and “playing pile” are descriptions of the notion of pile.

The terms “numeric value”, “ranks of the card”, “suit”, and “same rank” are properties of cards.

The nouns “number of cards” and “lack of cards” can be properties of a card deck or of a pile of cards.

A “turn” is an attribute that the game will keep track of as the playing proceeds.

A “tie” seems to be a property of a game.

The terms “clubs”, “diamonds”, “hearts”, “spades”, “ace”, “jack”, “queen”, and “king” are values of the properties suit and rank.

That leaves two nouns from the list, “process” and “cards that have been played”.

The notion of repeating the process will be embodied in the operation that controls the playing of the game.

The “cards that have been played” refer neither to a player’s playing pile nor to the pile of cards won. These cards are the ones placed on the table when a tie occurs, a situation that is call a “war” in the game. We may be talking about some other kind of pile, but let us postpone this issue till later.

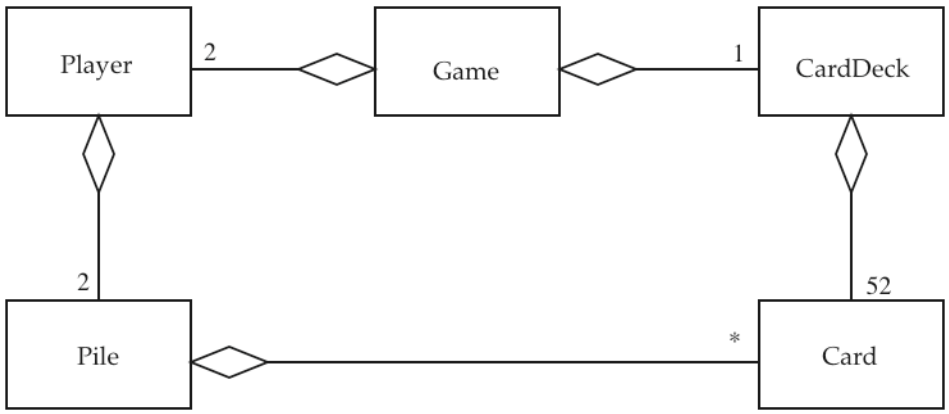
Relationships between Concepts

- A game has two players and a deck of cards.
- A player has two piles of cards.
- A deck has 52 cards.
- A pile has a bunch of cards.

Notice that no “is-a” relationships appear in this analysis.

All of the relations take the form of “has-a” or “has-some”, which means we will assemble the solution using composition of objects and not inheritance.

The following diagram shows the concepts and their relationships that we have discovered so far.



Behavior

We discover at least some of the operations needed in the solution by analyzing the verbs that occur in the specification.

play the game
shuffle the deck
deal
show cards
win cards
place number of cards
compare cards
win
tie occurs
repeat process
player finishes
are turned over
cannot complete turn
win the game

The main task in describing the behavior that we need comes in the assigning of operations to the concepts that we have identified.

Which objects will be responsible for which behavior?

Below we list each of the concepts, the candidates for classes, and try to place the verbs according to our analysis of responsibilities.

Card
    compare cards

CardDeck
    shuffle the deck
    deal

Player

- show cards
- win cards
- place number of cards

Pile

- are turned over

Game

- play the game
- repeat process
- win the game

Unassigned (deal with these verbs later)

- tie occurs
- cannot complete turn
- player finishes

At this point we move from the analysis phase to the design phase, although the boundary between the two can be a bit ambiguous.

We are ready, however, to start making some decisions about classes, their methods, and their properties.

Design of Card

The card concept clearly defines a kind of object, which can be modeled by a class we call Card.

So far the only behavior that we have identified is the ability to compare the values on two cards.

To integrate the Card class into the world of Java, we define Card to implement the interface Comparable and supply a *compareTo* method.

To enable the program to display the progress of a game of War we need a way to exhibit each card. We provide this behavior by overriding the *toString* method from Object.

The Card objects that the program creates will be immutable objects, so we have no need for mutator methods.

But accessor methods seem appropriate. Call them *getValue* and *getSuit*, since a play card is determined by a value and a suit.

A constructor uses these two pieces of information to create a card.

Implementation of Card

A Card object is defined by its value (or rank) and its suit.

We represent the value using the integers from 1 to 13.

Although the suit can be represented as a String object, we code the suits with the integer values 1, 2, 3, and 4.

The class definition is shown below. We add an *equals* method that is compatible with *compareTo*.


```
class Card implements Comparable
{
    Card(int r, int s)
    {
        rank = r;
        suit = s;
    }

    int getRank()
    {
        return rank;
    }

    int getSuit()
    {
        return suit;
    }

    public int compareTo(Object ob)
    {
        Card other = (Card)ob;
        int thisRank = this.getRank();
        int otherRank = other.getRank();
        if (thisRank == 1) thisRank = 14;    // make aces high
        if (otherRank == 1) otherRank = 14;
        return thisRank - otherRank;
    }

    public boolean equals(Object ob)
    {
        if (ob instanceof Card)
        {
            Card other = (Card)ob;
            return value==other.value && suit==other.suit;
        }
        else return false;
    }
```

```
    public String toString()
    {
        String val;
        String [] suitList =
            { "", "Clubs", "Diamonds", "Hearts", "Spades" };

        if (rank == 1) val = "Ace";
        else if (rank == 11) val = "Jack";
        else if (rank == 12) val = "Queen";
        else if (rank == 13) val = "King";
        else val = String.valueOf(rank);        // int to String

        String s = val + " of " + suitList[suit];

        for (int k=s.length()+1; k<=17; k++)   s = s + " ";

        return s;
    }

    private int rank;
    private int suit;
}
```

toString produces a string of the form "10 of Clubs".

Since these strings will have various lengths depending on the value and the suit, add spaces to bring the length up to 17 characters, the width of the longest string, "Queen of Diamonds".

Design of CardDeck

A CardDeck object will contain an array of 52 Card objects, which are created by the Card constructor, whose parameters are an integer value and an integer code representing the suit.

Since the cards will be dealt during the game, we maintain a property (an instance variable) to hold the number of cards left in the deck. That number counts down from 52 to 0 as the cards are dealt.

The *deal* method returns cards from the array starting at position 51. Since the method promises to return an object, we have it return **null** if it is called with an empty deck.

A CardDeck object responds to a method *getSize* that returns the number of card left in the deck as it is dealt.

Implementation of CardDeck

The job of creating the 52 cards is relegated to a private method *fill* that uses nested for loops to generate the 13 values and 4 suits.

```
class CardDeck
{
    CardDeck()
    {
        deck = new Card [52];
        fill();
    }
```

```
void shuffle()
{
    for (int next = 0; next < numCards-1; next++)
    {
        int r = myRandom(next, numCards-1);
        Card temp = deck[next];
        deck[next] = deck[r];
        deck[r] = temp;
    }
}

Card deal()
{
    if (numCards == 0) return null;
    numCards--;
    return deck[numCards];
}

int getSize()
{
    return numCards;
}

private void fill()
{
    int index = 0;
    for (int r = 1; r <= 13; r++)
        for (int s = 1; s <= 4; s++)
        {
            deck[index] = new Card(r, s);

            index++;
        }
    numCards = 52;
}
```



```
    private static int myRandom(int low, int high)
    {
        return (int)((high+1-low)*Math.random()+low);
    }

    private Card [] deck;
    private int numCards;
}
```

Design of Game, Player, and Pile

These three concepts are not as easily separated as Card and CardDeck.

We need to understand the playing of a turn to determine the behavior required of the Player objects and the Pile objects.

In fact, we need to specify the actions in a turn to be certain that we have all the behavior that will be required.

So at this point we step aside from the object-oriented design process and enter the world of procedural programming to describe the algorithm embodied in the playing of the game. This behavior will belong to an instance method of Game, called *play*.

Since a game could conceivably go on forever, we restrict a game of War to at most 100 turns.

The algorithm for the game is described as a sequence of steps.

1. Create a deck of cards.

2. Shuffle the deck of cards.

3. Create two players named Ernie and Burt.

4. Deal all of the cards to the two players.

5. Manage the turns, stopping when one player has insufficient cards to continue or after 100 turns.

   One turn proceeds as follows.

   5.1 Ensure that both players have at least one card. Otherwise stop the game.

   5.2 Have each player produce one card and compare the two cards.

   5.3 If one card is higher in rank, give the cards to the player with the higher card and end the turn.

   5.4 If the cards are of the same rank, we have a war.

      5.4.1 Check that each player has enough cards to continue. Otherwise end the game.

      5.4.2 Each player produces a number of cards equal to the rank of cards that caused the tie.

      5.4.3 If the last cards played are of different rank, give all the cards played in this turn to the winning player and end the turn.

      5.4.4 If the last cards played are of equal rank, go back to step 5.4.1.

When the game has finished, we need to know the winner.

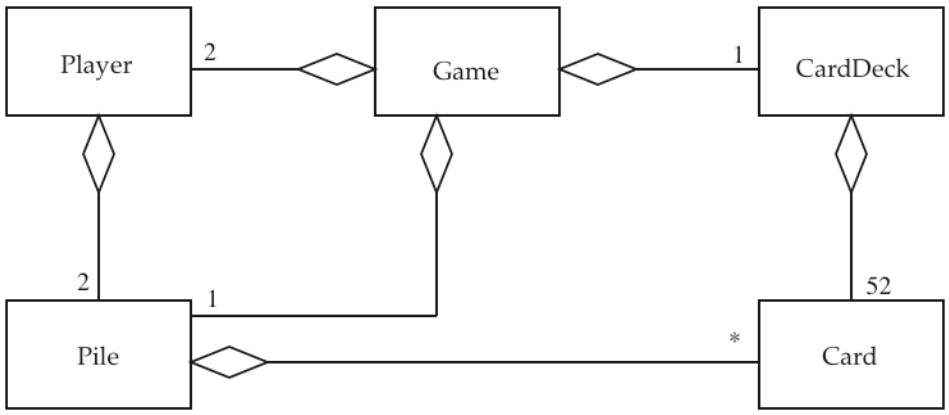
An instance method *getWinner* for the Game object returns the Player object for the winner, the player with the most cards.

In the both players finish with the same number of cards, *getWinner* returns the value **null**.

After developing the algorithm for play, we have a better understanding of the behavior required of Player and Pile.

Sometimes we add two cards one at a time to the pile of won cards, and sometimes we add a possibly large group of cards after a war.

To assemble these groups of cards, we give the Game object a pile of its own to collect the cards played in a war. The class diagram needs to be revised.



Player objects and Pile objects will have two methods for adding cards, one for individual cards when no tie occurs and one to add a Pile object after a war.

As a summary we list the methods needed in the three classes being designed.

Pile Methods

```
int getSize()              // number of cards in this pile
void clear()               // initialize this pile (to empty)
void addCard(Card c)       // add one card to this pile
void addCards(Pile p)      // add group of cards to this pile
Card nextCard()            // play a card from this pile
```

Player Methods

```
int numCards()       // number of cards held by this player
void collectCard(Card c)      // collect one card
void collectCards(Pile p)     // collect a group of cards
Card playCard()               // play one card
void useWonPile()       // transfer cards won to playing pile
String getName()        // return the name of player
```

Game Methods

```
void play()             // play the game
Player getWinner()      // return winning player or null
boolean enoughCards(int n)
                // true if both players have at least n cards
```

The last method tells whether both players have enough cards, the parameter, to continue the game.

Implementation of Game, Player, and Pile

A Pile object holds an array to contain its cards.

Two instance variables indicate the portion of the array that contains the cards belonging to the pile.

end is the position into which the next card is added to the pile.

front is the position from which the next card will be taken provided *front* < *end*.

A pile is empty if *front* has the same value as *end*.


```
class Pile
{
    Pile()
    {
        pile = new Card[52];
        front = 0;  end = 0;
    }

    int getSize()
    {
        return end - front;
    }

    void clear()
    {
         front = 0;  end = 0;
    }

    void addCard(Card c)
    {
        pile[end] = c;
        end++;
    }

    void addCards(Pile p)
    {
        while (p.getSize() > 0)
            addCard(p.nextCard());
    }
```

```
    Card nextCard()
    {
        if (front == end)
            return null;            // should not happen
        Card c = pile[front];
        front++;
        return c;
    }

    private Card [] pile;
    private int front, end;         // front ≤ end
}
```

The only methods that need explanation in the Player class are *playCard* and *useWonPile*.

If the player has no cards at all, an outcome that should not happen if the game is played correctly, the method *playCard* returns **null**.

If a player finds his or her playing pile is empty, the pile of cards won by that player becomes a new playing pile.

The *useWonPile* method initializes the playing pile so that its position variables, *front* and *end*, will not exceed its array size, adds the cards from the pile of won cards, and finally initializes (empties) that pile.


```
class Player
{
    Player(String n)
    {
        name = n;
        playPile = new Pile();
        wonPile = new Pile();
    }

    Card playCard()
    {
        if (playPile.getSize() == 0)
            useWonPile();
        if (playPile.getSize() > 0)
            return playPile.nextCard();
        return null;
    }

    String getName()
    {
        return name;
    }

    void collectCard(Card c)
    {
        wonPile.addCard(c);
    }

    void collectCards(Pile p)
    {
        wonPile.addCards(p);
    }
```

```
    void useWonPile()
    {
        playPile.clear();       // reset front and end to 0
        playPile.addCards(wonPile);
        wonPile.clear();        // reset front and end to 0
    }

    int numCards()
    {
        return playPile.getSize() + wonPile.getSize();
    }

    private Pile playPile, wonPile;
    private String name;
}
```

Since the card deck and the pile in the Game class are only used in the *play* method, they can be local variables.

The *play* method follows the outline of the algorithm given earlier.

The cards from the card deck are dealt into the “won” piles of the two players, after which these piles are turned into the playing piles for the players.

```
class Game
{
    void play()
    {
        CardDeck cd = new CardDeck();
        cd.shuffle();
        p1 = new Player("Ernie");
        p2 = new Player("Burt");
```



```
        while (cd.getSize() >= 2)
        {
            p1.collectCard(cd.deal());
            p2.collectCard(cd.deal());
        }
        p1.useWonPile();
        p2.useWonPile();
        Pile down = new Pile();    // Pile for cards in a war
loop:   for (int t=1; t<=100; t++)
        {
            if (!enoughCards(1)) break loop;
            Card c1 = p1.playCard();
            Card c2 = p2.playCard();
            System.out.println("\nTurn " + t + ": ");
            System.out.print(p1.getName() + ": " + c1 + " ");
            System.out.print(p2.getName() + ": " + c2 + " ");
            if (c1.compareTo(c2) > 0)
            {
                p1.collectCard(c1);      p1.collectCard(c2);
            }
            else if (c1.compareTo(c2) < 0)
            {
                p2.collectCard(c1);      p2.collectCard(c2);
            }
            else            // War
            {
                down.clear();
                down.addCard(c1);      down.addCard(c2);
                boolean done = false;
                do
                {   int num = c1.getRank();
                    if (!enoughCards(num)) break loop;
```

```
                System.out.print("\nWar! Players put down ");
                System.out.println(num + " card(s).");

                for (int m=1; m<=num; m++)
                {
                    c1 = p1.playCard(); c2 = p2.playCard();
                    down.addCard(c1);
                    down.addCard(c2);
                }
                System.out.print(p1.getName()+": "+ c1 + " ");
                System.out.print(p2.getName()+": " + c2 + " ");

                if (c1.compareTo(c2) > 0)
                {   p1.collectCards(down);
                    done = true;
                }
                else if (c1.compareTo(c2) < 0)
                {   p2.collectCards(down);
                    done = true;
                }
            }
            while (!done);
        }       // end of for t=1 to 100
        System.out.println(p1.numCards() + " to "
                                        + p2.numCards());
    }
}

boolean enoughCards(int n)
{
    if (p1.numCards() < n || p2.numCards() < n)
        return false;
    return true;
}
```



```
    Player getWinner()
    {
        if (p1.numCards() > p2.numCards())
            return p1;
        else if (p2.numCards() > p1.numCards())
            return p2;
        else
            return null;
    }

    private Player p1, p2;
}
```

Finally, we need a class to instantiate a Game object, call its *play* method, and report who won the game.

```
public class War
{
    public static void main(String [] args)
    {
        Game g = new Game();
        g.play();
        Player winner = g.getWinner();
        if (winner == null) System.out.println("Tie game.");
        else System.out.println("\nWinner = "
                                        + winner.getName());
    }
}
```

Default Packages

We have de-emphasized Java packages because more important topics involving programming and object-oriented design needed to be covered.

We have relied on default packages to collect our classes and interfaces, making them visible to each other.

Java allows two ways to create a default, anonymous package, which we illustrate with the six classes that solve the War Game problem.

1. If we put all six classes into the same source file, called War.java, they form a package with no name.

   Since the class War has the main method, it must be declared public, but the other five classes can have no visibility modifier (they have package visibility).

2. To be able to execute a Java class, the current directory (folder) containing that class must be in the path for the Java class loader.

   Therefore, all classes in the current directory form an anonymous package.

   These classes may be public or package visible (no modifier), although any class with a main method that we plan to execute must be public.

Building a Named Package

Using default packages works fine for the small programs that we write as we learn Java, but any Java software that entails a large number of classes and interfaces that we expect to share with others should be inserted into a package that other users will import to get access to the software.

To illustrate the process of creating a package, we build a package, named *war*, containing the five classes that solve the problem.

The class that controls the program, War.class, will remain outside of the package.

First, we create a directory, called *war*, that resides in the same directory that contains War.java, and move the other five source files into it. Here is the directory structure.

```
War
  |  War.java
  |  war
       |  Card.java
       |  CardDeck.java
       |  Pile.java
       |  Player.java
       |  Game.java
```

Before we can compile these files, any classes and methods accessed from War.java must be made public.

That means the classes Player and Game and the methods *play*, *getWinner*, and *getName* must have a **public** modifier.

In fact we may choose to export all five of the classes and most of their methods.

We can do this by placing **public** on the class headers and the method headers.

The source files for each of the five classes that go into the package *war* must begin with a statement, ***package*** *war;*.

The source file for War.java must begin with ***import*** *war.*;*.

To compile the classes we need to be in the War directory.

All the classes in the package must be compiled using the path to the source files.

If you compile from a command line in your operating system, use these commands.

```
javac war/Card.java
javac war/CardDeck.java
javac war/Pile.java
javac war/Player.java
javac war/Game.java
javac War.java
```

The War class is executed as you would expect.

```
java War
```

But if one of the package classes has a main method that you want to execute, say a test routine in CardDeck that deals a few cards, execute it from the upper directory and give the fully qualified name of the class.

```
java war.CardDeck
```

