Chapter 7

A Design Method

Concepts:
- ▷ Signatures
- ▷ Interface design
- ▷ Abstract base classes

But, luckily, he kept his wits and his purple crayon.
—Crockett Johnson

THROUGHOUT THE REST of this book we consider a number of data structures–first from an abstract viewpoint and then as more detailed implementations. In the process of considering these implementations, we will use a simple design method that focuses on the staged development of interfaces and abstract base classes. Here, we outline the design method. In the first section we describe the process. The remaining sections are dedicated to developing several examples.

7.1 The Interface-Based Approach

As we have seen in our discussion of abstract data types, the public aspect of the data type—that part that users depend on—is almost entirely incorporated in the *interface*. In Java the interface is a formal feature of the language. Interfaces allow the programmer of an abstract data type to specify the *signatures* of each of the externally visible methods, as well as any constants that might be associated with implementations.

The development and adherence to the interface is the most important part of the development of an implementation. The process can be outlined as follows:

1. Design of the interface. The interface describes the common external features of all implementations.

2. An abstract implementation. The abstract implementation describes the common internal features of all implementations.

3. Extension of the abstract class. Each implementation suggests an independent approach to writing code that extends the abstract implementation and supports the required elements of the interface.

7.1.1 Design of the Interface

The designer first considers, in as much detail as possible, an abstract data structure's various internal *states*. For example, if we are to consider a `Vector` abstractly, we pull out a napkin, and draw an abstract `Vector` and we consider the various effects that `Vector` *operations* might have on its *structure*. In our napkin-as-design-tool strategy, we might find ourselves using ellipses to suggest that we expect the `Vector` to be unbounded in size; we might use outward arrows to suggest the effect of accessor methods or methods that remove values; and we might blot out old `Vector` values in favor of new values when mutators are used. The designer must develop, from these free-flowing abstract notions of a structure, a collection of precise notions of how structures are accessed and mutated. It is also important to understand which states of the abstract structure are valid, and how the various methods ensure that the structure moves from one valid state to the next.

Armed with this information, the designer develops the interface—the external description of how users of the structure can interact with it. This consists of

1. A list of constant (`final static`) values that help provide meaning to values associated with the abstract structure. For example, any models of an atom might provide constants for the masses of electrons, protons, and neutrons. Each of these constants is declared within the `atom` interface and becomes part of any implementation.

2. Implementation-independent publicly accessible methods that access or modify data. For example, if we described an interface for a time-of-day clock, methods for reading and setting the current time would be declared part of the public interface. A method that made the clock "tick," causing time to move forward, would not be declared as part of the interface because it would not be a feature visible to the user. From the user's standpoint, the clock ticks on its own. How the clock ticks is not an abstract feature of the clock.

3. If an interface appears to be a refinement of another interface, it is common practice to have the one `extend` the other. This is useful if the new objects should be usable wherever the previously declared values are used.

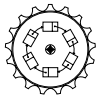
Once the interface has been defined, it can be put to immediate use. Potential users of the class can be asked to review it. In fact, application code can be written to make use of the new interface. Later, if particular implementations are developed, the application code can be put to use. It is important to remember, however, that the development of the application and the implementation of the data structure may both proceed at the same time.

7.1.2 Development of an Abstract Implementation

Once the interface has been outlined, it is useful for the designer to consider those functions of the structure that are implementation independent. These pieces are gathered together in a partial or *abstract* implementation of the interface called an *abstract base class*. Since some parts of the interface may not be implemented in the abstract base class—perhaps they require a commitment to a particular implementation approach—it is necessary for the class to be declared `abstract`. Once the abstract class is implemented, it may be used as a basis for developing extensions that describe particular implementations.

Some structures may have some built-in redundancy in their public methods. An interface supporting trigonometric calculations might, for example, have a method `tan` which computes its result from `sin` and `cos`. No matter how `sin` and `cos` are actually implemented, the `tan` method can be implemented in this manner. On the other hand, if a `tan` function can be computed in a more direct manner—one not dependent on `sin` and `cos`—a particular implementation might override the method outlined in the abstract base class.

A similar approach is taken in `Vector`-like classes that implement a backward-compatible `setElementAt` method based on the more recently added `set` method. Such a method might appear in an abstract base class as



Vector

```
public void setElementAt(E obj, int index)
// pre: 0 <= index && index < size()
// post: element value is changed to obj
{
    set(index,obj);
}
```

Because such code cannot be included in any interface that might describe a `Vector`, we place the code in the abstract class.

Implementors of other abstract objects may find it useful to develop a common library of methods that support all implementations. These methods—often declared privately—are only made available to the implementor of the class. If they are utility methods (like `sin` and `sqrt`) that are not associated with a particular object, we also declare them `static`.

Thus the abstract base class provides the basis for all implementations of an interface. This includes implementation-independent code that need not be rewritten for each implementation. Even if an abstract base class is empty, it is useful to declare the class to facilitate the implementation of implementation-independent development later in the life of the data structure. It is frequently the case, for example, that code that appears in several implementations is removed from the specific implementations and shared from the abstract base class. When implementations are actually developed, they `extend` the associated abstract base class.

7.1.3 Implementation

When the abstract base class is finished, it is then possible for one or more implementations to be developed. Usually each of these implementations extends the work that was started in the abstract base class. For example, a `Fraction` interface might be implemented as the ratio of two values (as in the `Ratio` class we saw in Chapter 1), or it might be implemented as a `double`. In the latter case, the double is converted to an approximate ratio of two integers, the numerator and denominator. In both cases, it might be useful to declare an `AbstractFraction` class that includes a greatest common divisor (`gcd`) method. Such a method would be declared `protected` and `static`.

7.2 Example: Development of Generators

As an example, we develop several implementations of an object that generates a sequence of values—an object we will call a *generator*. Such an object might generate randomly selected values, or it might generate, incrementally, a sequence of primes.

We imagine the interface would include the following methods:

Generator

```
public interface Generator
{
    public void reset();
    // post: the generator is reset to the beginning of the sequence;
    //   the current value can be immediately retrieved with get.

    public int next();
    // post: returns true iff more elements are to be generated.

    public int get();
    // post: returns the current value of the generator.
}
```

The `Generator` is constructed and the `get` method can be used to get its initial value. When necessary, the `next` routine generates, one at a time, a sequence of integer values. Each call to `next` returns the next value in the sequence, a value that can be recalled using the `get` method. If necessary, the `reset` method can be used to restart the sequence of generated values.

The next step is to generate an abstract base class that implements the interface. For the `AbstractGenerator` we implement any of the methods that can be implemented in a general manner. We choose, here, for example, to provide a mechanism for the `next` method to save the current value:

Abstract-
Generator

```
abstract public class AbstractGenerator
    implements Generator
{
    protected int current;  // the last value saved
```

```
    public AbstractGenerator(int initial)
    // post: initialize the current value to initial
    {
        current = initial;
    }

    public AbstractGenerator()
    // post: initialize the current value to zero
    {
        this(0);
    }

    protected int set(Integer next)
    // post: sets the current value to next, and extends the sequence
    {
        int result = current;
        current = next;
        return result;
    }

    public int get()
    // post: returns the current value of the sequence
    {
        return current;
    }

    public void reset()
    // post: resets the Generator (by default, does nothing)
    {
    }
}
```

The `current` variable keeps track of a single integer—ideally the last value generated. This value is the value returned by the `get` method. A hidden method—`set`—allows any implementation to set the value of `current`. It is expected that this is called by the `next` method of the particular implementation. By providing this code in the abstract base class, individual implementations needn't repeatedly reimplement this common code. By default, the `reset` method does nothing. If a particular generator does not require a `reset` method, the default method does nothing.

Here is a simple implementation of a `Generator` that generates a constant value. The value is provided in the constructor:

`Constant-Generator`

```
public class ConstantGenerator extends AbstractGenerator
{
    public ConstantGenerator(int c)
    // pre: c is the value to be generated.
    // post: constructs a generator that returns a constant value
```

```
    {
        set(c);
    }

    public int next()
    // post: returns the constant value
    {
        return get();
    }
}
```

The `set` method of the `AbstractGenerator` is called from the constructor, recording the constant value to be returned. This effectively implements the `get` method—that code was provided in the `AbstractGenerator` class. The `next` method simply returns the value available from the `get` method.

Another implementation of a `Generator` returns a sequence of prime numbers. In this case, the constructor sets the current value of the generator to 2—the first prime. The `next` method searches for the next prime and returns that value after it has been saved. Here, the private `set` and the public `get` methods from the `AbstractGenerator` class help to develop the state of the `Generator`:

PrimeGenerator

```
public class PrimeGenerator extends AbstractGenerator
{
    public PrimeGenerator()
    // post: construct a generator that delivers primes starting at 2.
    {
        reset();
    }

    public void reset()
    // post: reset the generator to return primes starting at 2
    {
        set(2);
    }

    public int next()
    // post: generate the next prime
    {
        int f,n = get();
        do
        {
            if (n == 2) n = 3;
            else n += 2;

            // check the next value
            for (f = 2; f*f <= n; f++)
            {
                if (0 ==(n % f)) break;
```

```
            }
        } while (f*f <= n);
        set(n);
        return n;
    }
}
```

Clearly, the `reset` method is responsible for restarting the sequence at 2. While it would be possible for each `Generator` to keep track of its current value in its own manner, placing that general code in the `AbstractGenerator` reduces the overall cost of keeping track of this information for each of the many implementations.

Exercise 7.1 *Implement a* `Generator` *that provides a stream of random integers. After a call to* `reset`*, the random sequence is "rewound" to return the same sequence again. Different generators should probably generate different sequences. (Hint: You may find it useful to use the* `setSeed` *method of* `java.util.Random`*.)*

7.3 Example: Playing Cards

Many games involve the use of playing cards. Not all decks of cards are the same. For example, a deck of bridge cards has four suits with thirteen cards in each suit. There are no jokers. Poker has a similar deck of cards, but various games include special joker or wild cards. A pinochle deck has 48 cards consisting of two copies of 9, jack, queen, king, 10, and ace in each of four suits. The ranking of cards places 10 between king and ace. A baccarat deck is as in bridge, except that face cards are worth nothing. Cribbage uses a standard deck of cards, with aces low.

While there are many differences among these card decks, there are some common features. In each, cards have a suit and a face (e.g., ace, king, 5). Most decks assign a value or rank to each of the cards. In many games it is desirable to be able to compare the relative values of two cards. With these features in mind, we can develop the following `Card` interface:

`Card`

```
public interface Card
{
    public static final int ACE = 1;
    public static final int JACK = 11;
    public static final int QUEEN = 12;
    public static final int KING = 13;
    public static final int JOKER = 14;
    public static final int CLUBS = 0;
    public static final int DIAMONDS = 1;
    public static final int HEARTS = 2;
    public static final int SPADES = 3;
    public int suit();
```

```
    // post: returns the suit of the card

    public int face();
    // post: returns the face of the card, e.g., ACE, 3, JACK

    public boolean isWild();
    // post: returns true iff this card is a wild card

    public int value();
    // post: return the point value of the card

    public int compareTo(Object other);
    // pre: other is valid Card
    // post: returns int <,==,> 0 if this card is <,==,> other

    public String toString();
    // post: returns a printable version of this card
}
```

The card interface provides all the public methods that we need to have in our card games, but it does not provide any hints at how the cards are implemented. The interface also provides standard names for faces and suits that are passed to and returned from the various card methods.

In the expectation that most card implementations are similar to a standard deck of cards, we provide an `AbstractCard` class that keeps track of an integer—a card index—that may be changed with `set` or retrieved with `get` (both are protected methods):

`AbstractCard`

```
import java.util.Random;
public abstract class AbstractCard implements Card
{
    protected int cardIndex;
    protected static Random gen = new Random();

    public AbstractCard()
    // post: constructs a random card in a standard deck
    {
        set(randomIndex(52));
    }

    protected static int randomIndex(int max)
    // pre: max > 0
    // post: returns a random number n, 0 <= n < max
    {
        return Math.abs(gen.nextInt()) % max;
    }

    protected void set(int index)
    // post: this card has cardIndex index
    {
```

```
    cardIndex = index;
}

protected int get()
// post: returns this card's card index
{
    return cardIndex;
}

public int suit()
// post: returns the suit of the card
{
    return cardIndex / 13;
}

public int face()
// post: returns the face of the card, e.g. ACE, 3, JACK
{
    return (cardIndex % 13)+1;
}

public boolean isWild()
// post: returns true iff this card is a wild card
// (default is false)
{
    return false;
}

public int value()
// post: return the point value of the card, Ace..King
{
    return face();
}

public String toString()
// post: returns a printable version of this card
{
    String cardName = "";
    switch (face())
    {
      case ACE: cardName = "Ace"; break;
      case JACK: cardName = "Jack"; break;
      case QUEEN: cardName = "Queen"; break;
      case KING: cardName = "King"; break;
      default: cardName = cardName + face(); break;
    }
    switch (suit())
    {
      case HEARTS: cardName += " of Hearts"; break;
      case DIAMONDS: cardName += " of Diamonds"; break;
```

```
            case CLUBS: cardName += " of Clubs"; break;
            case SPADES: cardName += " of Spades"; break;
        }
        return cardName;
    }
}
```

Our abstract base class also provides a protected random number generator that returns values from 0 to `max`-1. We make use of this in the default constructor for a standard deck; it picks a random card from the usual 52. The cards are indexed from the ace of clubs through the king of spades. Thus, the `face` and `suit` methods must use division and modulo operators to split the card index into the two constituent parts. By default, the `value` method returns the face of the card as its value. This is likely to be different for different implementations of cards, as the face values of cards in different games vary considerably.

We also provide a standard `toString` method that allows us to easily print out a card. We do not provide a `compareTo` method because there are complexities with comparing cards that cannot be predicted at this stage. For example, in bridge, suits play a predominant role in comparing cards. In baccarat they do not.

Since a poker deck is very similar to our standard implementation, we find the `PokerCard` implementation is very short. All that is important is that we allow aces to have high values:

PokerCard

```
public class PokerCard extends AbstractCard
{
    public PokerCard(int face, int suit)
    // pre: face and suit have valid values
    // post: constructs a card with the particular face value
    {
        set(suit*13+face-1);
    }

    public PokerCard()
    // post: construct a random poker card.
    {
        // by default, calls the AbstractCard constructor
    }

    public int value()
    // post: returns rank of card - aces are high
    {
        if (face() == ACE) return KING+1;
        else return face();
    }

    public int compareTo(Object other)
    // pre: other is valid PokerCard
    // post: returns relationship between this card and other
```

```
    {
        PokerCard that = (PokerCard)other;
        return value()-that.value();
    }
}
```

Exercise 7.2 *Write the* `value` *and* `compareTo` *methods for a pair of cards where suits play an important role. Aces are high, and assume that suits are ranked clubs (low), diamonds, hearts, and spades (high). Assume that face values are only considered if the suits are the same; otherwise ranking of cards depends on their suits alone.*

The implementation of a pinochle card is particularly difficult. We are interested in providing the standard interface for a pinochle card, but we are faced with the fact that there are two copies each of the six cards 9, jack, queen, king, 10, and ace, in each of the four suits. Furthermore we assume that 10 has the unusual ranking between king and ace. Here's one approach:

PinochleCard

```
public class PinochleCard extends AbstractCard
{
    // cardIndex        face    suit
    // 0                9       clubs
    // 1                9       clubs (duplicate)
    // ...
    // 10               ACE     clubs
    // 11               ACE     clubs (duplicate)
    // 12               9       diamonds
    // 13               9       diamonds (duplicate)
    // ...
    // 47               ACE     spades (duplicate)

    public PinochleCard(int face, int suit, int copy)
    // pre: face and suit have valid values
    // post: constructs a card with the particular face value
    {
        if (face == ACE) face = KING+1;
        set((suit*2+copy)*6+face-9);
    }

    public PinochleCard()
    // post: construct a random Pinochle card.
    {
        set(randomIndex(48));
    }

    public int face()
    // post: returns the face value of the card (9 thru Ace)
    {
        int result = get()%6 + 9;
        if (result == 14) result = ACE;
```

```
            return result;
        }

        public int suit()
        // post: returns the suit of the card (there are duplicates!)
        {
            // this is tricky; we divide by 12 cards (including duplicates)
            // per suit, and again by 2 to remove the duplicate
            return cardIndex / 12 / 2;
        }

        public int value()
        // post: returns rank of card - aces are high
        {
            if (face() == ACE) return KING+2;
            else if (face() == 10) return KING+1;
            else return face();
        }

        public int compareTo(Object other)
        // pre: other is valid PinochleCard
        // post: returns relationship between this card and other
        {
            PinochleCard that = (PinochleCard)other;
            return value()-that.value();
        }
    }
```

The difficulty is that there is more than one copy of a card. We choose to keep track of the extra copy, in case we need to distinguish between them at some point, but we treat duplicates the same in determining face value, suit, and relative rankings.

7.4 Conclusions

Throughout the remainder of this book we will find it useful to approach each type of data structure first in an abstract manner, and then provide the details of various implementations. While each implementation tends to have a distinct approach to supporting the abstract structure, many features are common to *all* implementations. The basic interface, for example, is a shared concept of the methods that are used to access the data structure. Other features—including common private methods and shared utility methods—are provided in a basic implementation called the *abstract base class*. This incomplete class serves as a single point of extension for many implementations; the public and private features of the abstract base class are shared (and possibly overridden) by the varied approaches to solving the problem.