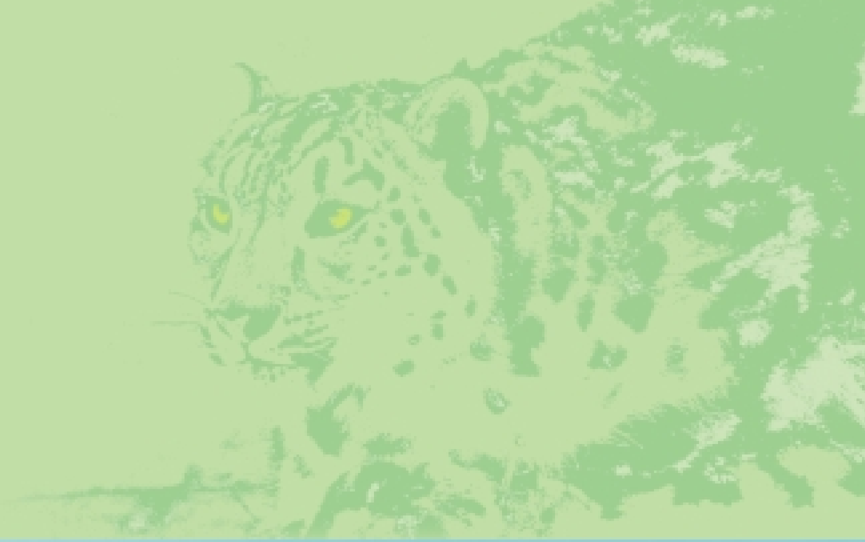


CHAPTER 12

UML and Patterns

12 UML and Patterns

Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity.

F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987

Introduction

UML and patterns are two software design tools that apply no matter what programming language you are using, as long as the language provides for classes and related facilities for object-oriented programming (OOP). This chapter presents a very brief introduction to these two topics. This chapter contains no new details about the Java language.

UML is a graphical language that is used for designing and documenting software created within the OOP framework.

A pattern in programming is very similar to a pattern in any other context. It is a kind of template or outline of a software task that can be realized as different code in different, but similar, applications.

Prerequisites

Section 12.1 on UML and Section 12.2 on patterns can be read in either order. None of this chapter is needed for the rest of the chapters in this book.

Section 12.1 on UML uses material from Chapters 1–5 and Chapter 7 on inheritance.

Section 12.2 on patterns uses material from Chapters 1–7 and Chapter 11.

12.1 UML

One picture is worth a thousand words.

Chinese proverb

Most people do not think in Java or in any other programming language. As a result, computer scientists have always sought to produce more human-oriented ways of representing programs. One widely used representation is pseudocode, which is a mixture of a programming language like Java and a natural language like English. To think about a programming problem without needing to worry about the syntax details of a language like Java, you can simply relax the syntax rules and write in pseudocode. Pseudocode has become a standard tool used by programmers, but pseudocode is a linear and algebraic representation of programming. Computer scientists have long sought to give software design a graphical representation. To this end, a number of graphical representation systems for program design have been proposed, used, and ultimately found to be wanting. Terms like *flowchart*, *structure diagram*, and many more names of graphical program representations are today only recognized by those of the older generation. Today's candidate for a graphical representation formalism is the **Unified Modeling Language** (or **UML**). UML was designed to reflect and be used with the OOP philosophy. It is too early to say whether or not UML will stand the test of time, but it is off to a good start. A number of companies have adopted the UML formalism to use in their software design projects.

UML

HISTORY OF UML

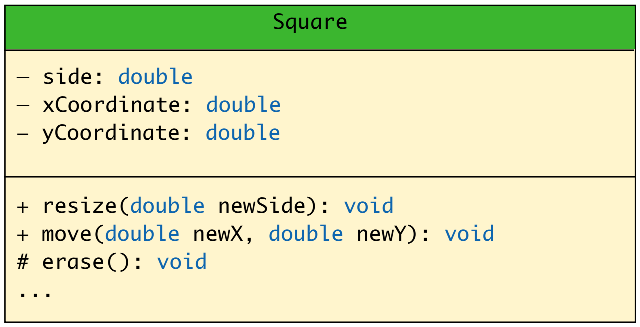
UML developed along with OOP. As the OOP philosophy became more and more commonly used, different groups developed their own graphical or other representations for OOP design. In 1996, Grady Booch, Ivar Jacobson, and James Rumbaugh released an early version of UML. UML was intended to bring together the various different graphical representation methods to produce a standardized graphical representation language for object-oriented design and documentation. Since that time, UML has been developed and revised in response to feedback from the OOP community. Today the UML standard is maintained and certified by the Object Management Group (OMG), a nonprofit organization that promotes the use of object-oriented techniques.

UML CLASS DIAGRAMS

Classes are central to OOP and the **class diagram** is the easiest of the UML graphical representations to understand and use. Display 12.1 shows the class diagram for a class to represent a square. The diagram consists of a box divided into three sections. (The colors are optional and not standardized.) The top section has the class name, `Square`. The next section has the data specification for the class. In this example there are three pieces of data (three instance variables), a value of type `double` giving the length of a side, and two more values of type `double` giving the *x* and *y* coordinates of the center of the square. The third section gives the actions (class methods). The notation for method entries is not identical to that of a Java method heading but it contains the same information. A minus sign indicates a private member. So, for the class `Square`, all data is private. A plus sign indicates a public member. A sharp (#) indicates a protected member. A tilde (~) indicates package access. So, for the class `Square`, the class diagram shows two public methods and one protected method. A class diagram need not give a complete description of the class. When you do not need all the members in a class for the analysis at hand, you do not list all the members in the class diagram. Missing members are indicated with an ellipsis (three dots).

class diagram

Display 12.1 A UML Class Diagram



```
Square
– side: double
– xCoordinate: double
– yCoordinate: double

+ resize(double newSide): void
+ move(double newX, double newY): void
# erase(): void
...
```

CLASS INTERACTIONS

Class diagrams by themselves are of little value, since they simply repeat the class interface, possibly with ellipses. To understand a design, you need to indicate how objects of the various classes interact. UML has various ways to indicate class interactions; for example, various sorts of annotated arrows indicate the information flow from one class object to another. UML also has annotations for class groupings into packages, annotations for inheritance, and annotations for other interactions. Moreover, UML is extensible. If what you want and need is not in UML, you can add it to UML. Of course, this all takes place inside a prescribed framework so that different software developers can understand each other's UML. One of the most fundamental of class interactions is inheritance, which is discussed in the next subsection.

INHERITANCE DIAGRAMS

inheritance diagram

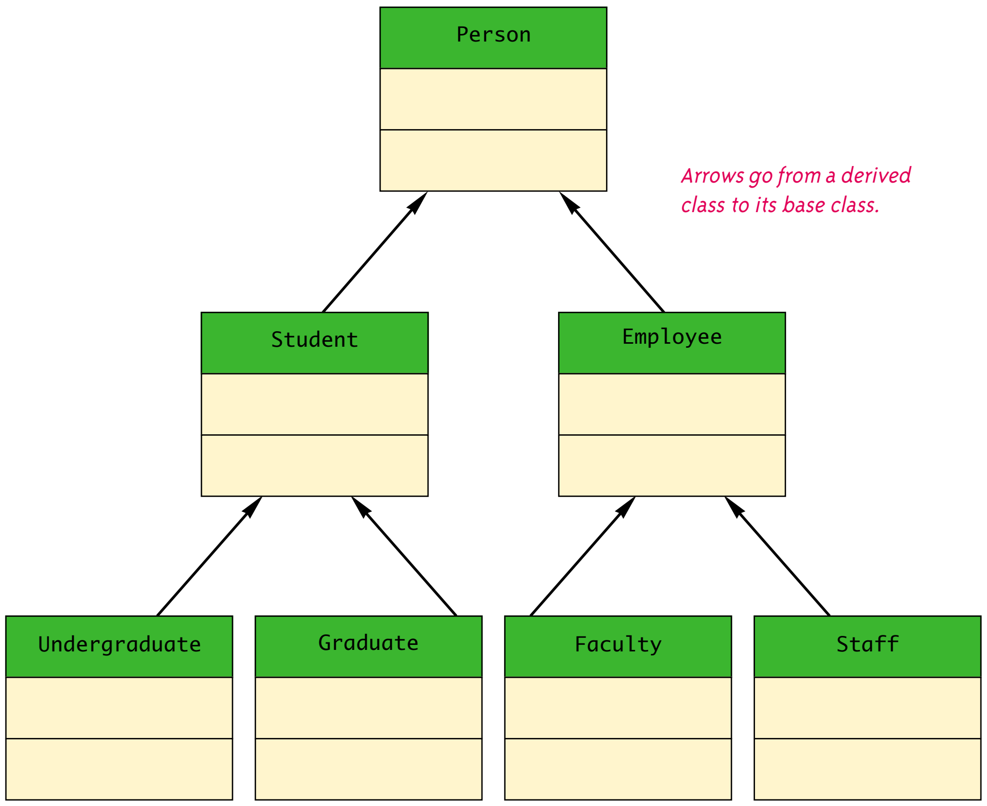
Display 12.2 shows a possible **inheritance diagram** for some classes used in a university's record-keeping software. Note that the class diagrams are incomplete. You normally show only as much of the class diagram as you need for the design task at hand. Note that the arrow heads point up from a derived class to its base class.

arrows

The arrows also help in locating method definitions. If you are looking for a method definition for some class, the arrows show the path you (or the computer) should follow. If you are looking for the definition of a method used by an object of the class `Undergraduate`, you first look in the definition of the class `Undergraduate`; if it is not there, you look in the definition of `Student`; if it is not there, you look in the definition of the class `Person`.

Display 12.3 shows some possible additional details of the inheritance hierarchy for the two classes `Person` and one of its derived classes, `Student`. Suppose `s` is an object of

Display 12.2 **A Class Hierarchy in UML Notation**



the class `Student`. The diagram in Display 12.3 tells you that you can find the definition of

```
s.toString();
```

and

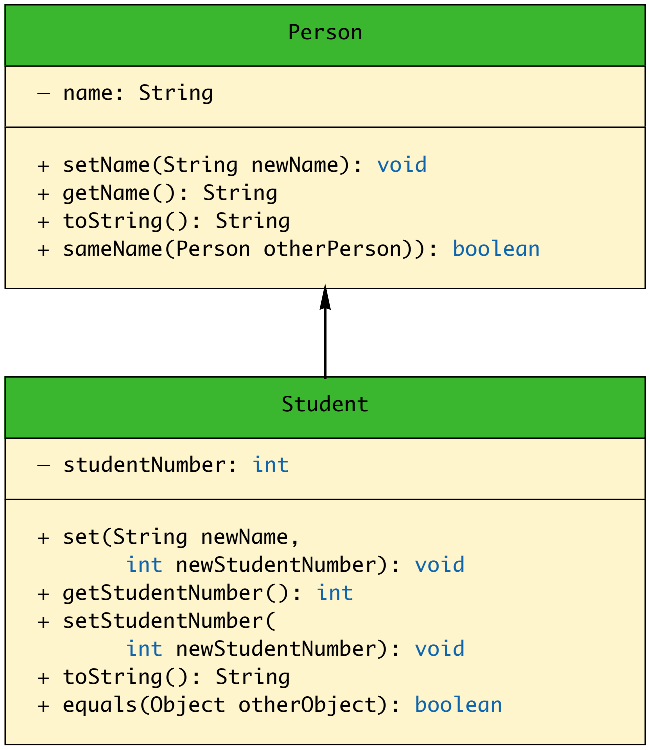
```
s.set("Joe Student", 4242);
```

in the class `Student`, but the definition of

```
s.setName("Josephine Student");
```

is found in the definition of the class `Person`.

Display 12.3 Some Details of a UML Class Hierarchy


Person
– name: String
+ setName(String newName): void
+ getName(): String
+ toString(): String
+ sameName(Person otherPerson)): boolean
Student
– studentNumber: int
+ set(String newName,
int newStudentNumber): void
+ getStudentNumber(): int
+ setStudentNumber(
int newStudentNumber): void
+ toString(): String
+ equals(Object otherObject): boolean

MORE UML

This is just a hint of what UML is all about. If you are interested in learning more, consult one of the many available references on UML.

Self-Test Exercises

1. Draw a class diagram for a class whose objects represent circles. Use Display 12.1 as a model.
2. Suppose `aStudent` is an object of the class `Student`. Based on the inheritance diagram in Display 12.3, where will you find the definition of the method `sameName` used in the following invocation, which compares `aStudent` and another object named `someStudent`? Explain your answer.

```
Student someStudent =
              new Student("Joe Student", 7777);
```

```
if (aStudent.sameName(someStudent))
    System.out.println("wow");
```

3. Suppose aStudent is an object of the class Student. Based on the inheritance diagram in Display 12.3, where will you find the definition of the method used in the following invocation? Explain your answer.

```
aStudent.setNumber(4242);
```

12.2 Patterns

I bid him look into the lives of men as though into a mirror,
and from others to take an example for himself.

Terence (Publius Terentius Afer) 190-159 B.C., *Adelphoe*

Patterns are design outlines that apply across a variety of software applications. To be useful, the pattern must apply across a variety of situations. To be substantive, the pattern must make some assumptions about the domain of applications to which it applies. For example, one well-known pattern is the **Container-Iterator** pattern. A **container** is a class (or other construct) whose objects hold multiple pieces of data. One example of a container is an array. Other examples, which will be discussed later in this book, are vectors and linked lists. Any class or other construct designed to hold multiple values can be viewed as a container. For example, a String value can be viewed as a container that contains the characters in the string. Any construct that allows you to cycle through all the items in a container is an **iterator**. For example, an array index is an iterator for an array. It can cycle through the array as follows:

pattern

Container-Iterator

container

iterator

```
for (int i; i < a.length; i++)
    Do something with a[i]
```

The index variable i is the iterator. The Container-Iterator pattern describes how an iterator is used on a container.

In this brief chapter we can give you only a taste of what patterns are all about. In this section we will discuss a few sample patterns to let you see what patterns look like. There are many more known and used patterns and many more yet to be explicated. This is a new and still developing field of software engineering.

ADAPTOR PATTERN ✜

The **Adaptor** pattern transforms one class into a different class without changing the underlying class but merely by adding a new interface. (The new interface replaces the old interface of the underlying class.) For example, in Chapter 11 we mentioned the `stack` data structure, which is used to, among other things, keep track of recursion. One way to create a stack data structure is to start with an array and add the stack interface. The Adaptor pattern says start with a container, like an array, and add an interface, like the stack interface.

THE MODEL-VIEW-CONTROLLER PATTERN ✜

Model-View-Controller

The **Model-View-Controller** pattern is a way of separating the I/O task of an application from the rest of the application. The Model part of the pattern performs the heart of the application. The View part is the output part; it displays a picture of the Model's state. The Controller is the input part; it relays commands from the user to the Model. Normally, each of the three interacting parts is realized as an object with responsibilities for its own tasks. The Model-View-Controller pattern is an example of a divide-and-conquer strategy. One big task is divided into three smaller tasks with well-defined responsibilities. Display 12.4 gives a diagram of the Model-View-Controller pattern.

As a very simple example, the Model might be a container class, such as an array. The View might display one element of the array. The Controller gives commands to display the element at a specified index. The Model (the array) notifies the View to display a new element whenever the array contents change or a different index location is given.

Any application can be made to fit the Model-View-Controller pattern, but it is particularly well suited to GUI (Graphical User Interface) design projects where the View can indeed be a visualization of the state of the Model. (A GUI interface is simply a windowing interface of the form you find in most modern software applications, as opposed to the simple text I/O we have used so far in this book.) For example, the Model might be an object to represent your list of computer desktop object names. The View could then be a GUI object that produces a screen display of your desktop icons. The Controller relays commands to the Model (which is a desktop object) to add or delete names. The Model object notifies the View object when the screen needs to be updated.

We have presented the Model-View-Controller pattern as if the user were the Controller. That was done primarily to simplify the examples. The Controller need not be under the direct control of the user, but could be some other kind of software or hardware component.

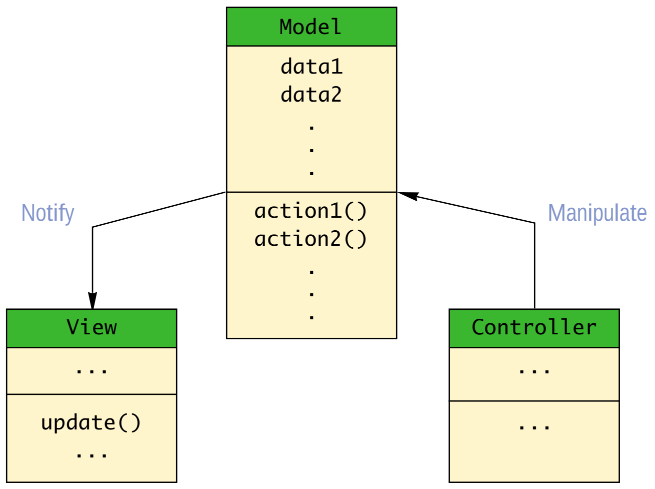
Example

A Sorting Pattern

The most efficient sorting algorithms all seem to follow a similar pattern. Expressed recursively, they divide the list of elements to be sorted into two smaller lists, recursively sort the two smaller lists, and then recombine the two sorted lists to obtain the final sorted list. In Display 12.5 this pattern is expressed as pseudocode (in fact, almost correct Java code) for a method to sort an array into increasing order using the < operator.

Our sorting pattern uses a divide-and-conquer strategy. It divides the entire collection of elements to be sorted into two smaller collections, sorts the smaller collections by recursive calls,

Display 12.4 **Model-View-Controller Pattern**



Display 12.5 **Divide-and-Conquer Sorting Pattern**

```
/**
 Precondition: Interval a[begin] through a[end] of a have elements.
 Postcondition: The values in the interval have
 been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
*/
public static void sort(Type[] a, int begin, int end)
{
    if ((end - begin) >= 1)
    {
        int splitPoint = split(a, begin, end);
        sort(a, begin, splitPoint);
        sort(a, splitPoint + 1, end);
        join(a, begin, splitPoint, end);
    }//else sorting one (or fewer) elements so do nothing.
}
```

To get a correct Java method definition Type must be replaced with a suitable type name.

Different definitions for the methods `split` and `join` will give different realizations of this pattern.

and then combines the two sorted collections to obtain the final sorted array. The following is the heart of our sorting pattern:

```
int splitPoint = split(a, begin, end);
sort(a, begin, splitPoint);
sort(a, splitPoint + 1, end);
join(a, begin, splitPoint, end);
```

Although the pattern does impose some minimum requirements on the methods `split` and `join`, the pattern does not say exactly how the methods `split` and `join` are defined. Different definitions of `split` and `join` will yield different sorting algorithms.

`split`

The method `split` rearranges the elements in the interval `a[begin]` through `a[end]` and divides the rearranged interval at a split point, `splitPoint`. The two smaller intervals `a[begin]` through `a[splitPoint]` and `[splitPoint + 1]` through `a[end]` are then sorted by a recursive call to the method `sort`. Note that the `split` method both rearranges the elements in the array interval `a[begin]` through `a[end]` and returns the index `splitPoint` that divides the interval. After the two smaller intervals are sorted, the method `join` then combines the two sorted intervals to obtain the final sorted version of the entire larger interval.

`join`

The pattern says nothing about how the method `split` rearranges and divides the interval `a[begin]` through `a[end]`. In a simple case, `split` might simply choose a value `splitPoint` between `begin` and `end` and divide the interval into the points before `splitPoint` and the points after `splitPoint`, with no rearranging. We will see an example that realizes the sorting pattern by defining `split` this way. On the other hand, the method `split` could do something more elaborate like move all the "small" elements to the front of the array and all the "large" elements toward the end of the array. This would be a step on the way to fully sorting the values. We will also see an example that realizes the sorting pattern in this second way.

merge sort

The simplest realization of this sorting pattern is the **merge sort** realization given in Display 12.6. In this realization the array base type, *Type*, is specialized to the type `double`. The merge sort is an example where the definition of `split` is very simple. It simply divides the array into two intervals with no rearranging of elements. The `join` method is more complicated. After the two subintervals are sorted, the method `join` merges the two sorted subintervals, copying elements from the array to a temporary array. The merging starts by comparing the smallest elements in each smaller sorted interval. The smaller of these two elements is the smallest of all the elements in either subinterval and so it is moved to the first position in the temporary array. The process is then repeated with the remaining elements in the two smaller sorted intervals to find the next smallest element, and so forth. A demonstration of using the merge sort version of `sort` is given in Display 12.7.

There is a trade-off between the complexity of the methods `split` and `join`. You can make either of them simple at the expense of making the other more complicated. For merge sort, `split` was simple and `join` was complicated. We next give a realization where `split` is complicated and `join` is simple.

quick sort

Display 12.8 gives the **quick sort** realization of our sorting pattern for the type `double`.

splitting value

In the quick sort realization, the definition of `split` is quite sophisticated. An arbitrary value in the array is chosen; this value is called the **splitting value**. In our realization, we chose `a[begin]` as the splitting value, but any value will do equally well. The elements in the array are rearranged so that all those elements that are less than or equal to the splitting value are at the front of the array, all the values that are greater than the splitting value are at the other end of the array, and the splitting value is placed so that it divides the entire array into these smaller and larger elements. Note that the smaller elements are not sorted and the larger elements are not sorted, but all the elements before the splitting value are smaller than any of the elements after the splitting value. The smaller elements are sorted by a recursive call, the larger elements are sorted by another recursive call, and then these two sorted segments are combined with the `join` method.

Display 12.6 Merge Sort Realization of Sorting Pattern *(Part 1 of 2)*

```java
/**
 Class that realizes the divide-and-conquer sorting pattern and
 uses the merge sort algorithm.
*/
public class MergeSort
{

    /**
     Precondition: Interval a[begin] through a[end] of a have elements.
     Postcondition: The values in the interval have
     been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
    */
    public static void sort(double[] a, int begin, int end)
    {
        if ((end - begin) >= 1)
        {
            int splitPoint = split(a, begin, end);
            sort(a, begin, splitPoint);
            sort(a, splitPoint + 1, end);
            join(a, begin, splitPoint, end);
        }//else sorting one (or fewer) elements so do nothing.
    }

    private static int split(double[] a, int begin, int end)
    {
        return ((begin + end)/2);
    }

    private static void join(double[] a, int begin, int splitPoint, int end)
    {
        double[] temp;
        int intervalSize = (end - begin + 1);
        temp = new double[intervalSize];
        int nextLeft = begin; //index for first chunk
        int nextRight = splitPoint + 1; //index for second chunk
        int i = 0; //index for temp

        //Merge till one side is exhausted:
        while ((nextLeft <= splitPoint) && (nextRight <= end))
        {
            if (a[nextLeft] < a[nextRight])
            {
                temp[i] = a[nextLeft];
                i++; nextLeft++;
            }
```

The method `sort` is identical to the version in the pattern (Display 12.5) except that Type is replaced with `double`.

Display 12.6 **Merge Sort Realization of Sorting Pattern *(Part 2 of 2)***

```
            else
            {
                temp[i] = a[nextRight];
                i++; nextRight++;
            }
        }

        while (nextLeft <= splitPoint)//Copy rest of left chunk, if any.
        {
            temp[i] = a[nextLeft];
            i++; nextLeft++;
        }

        while (nextRight <= end) //Copy rest of right chunk, if any.
        {
            temp[i] = a[nextRight];
            i++; nextRight++;
        }

        for (i = 0; i < intervalSize; i++)
            a[begin + i] = temp[i];
    }

}
```

In this case, the `join` method is as simple as it could be. It does nothing. Since the sorted smaller elements all precede the sorted larger elements, the entire array is sorted.

extra code on CD

A demonstration program for the quick sort method `sort` in Display 12.8 is given in the file `QuickSortDemo.java` on the accompanying CD.

(Both the merge sort and the quick sort realizations can be done without the use of a second temporary array, `temp`. However, that detail would only distract from the message of this example. In a real application, you may or may not, depending on details, want to consider the possibility of doing a sort realization without the use of the temporary array.)

RESTRICTIONS ON THE SORTING PATTERN

The sorting pattern, like all patterns, has some restrictions on where it applies. As we formulated the sorting pattern, it applies only to types for which the < operator is defined and it applies only to sorting into increasing order; it does not apply to sorting into decreasing order. However, this is a result of our simplifying details to make the

Display 12.7 Using the MergeSort Class

```
public class MergeSortDemo
{
    public static void main(String[] args)
    {
        double[] b = {7.7, 5.5, 11, 3, 16, 4.4, 20, 14, 13, 42};

        System.out.println("Array contents before sorting:");
        int i;
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();

        MergeSort.sort(b, 0, b.length-1);
        System.out.println("Sorted array values:");
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
    }
}
```

SAMPLE DIALOGUE

```
Array contents before sorting:
7.7 5.5 11.0 3.0 16.0 4.4 20.0 14.0 13.0 42.0
Sorted array values:
3.0 4.4 5.5 7.7 11.0 13.0 14.0 16.0 20.0 42.0
```

presentation clearer. You can make the pattern more general by replacing the < operator with a `boolean` valued method called `compare` that has two arguments of the base type of the array and that returns `true` or `false` depending on whether the first "comes before" the second. Then, the only restriction is that the `compare` method must have a reasonable definition.[1] This sort of generalization is discussed in Chapter 13 in the subsection entitled "The `Comparable` Interface."

[1] The technical requirement is that the `compare` method be a *total ordering*, a concept discussed in Chapter 13. Essentially, all common orderings that you might want to sort by are total orderings.

The `Comparable` interface has a method `compareTo`, which is slightly different from `compare`. However, the method we described as `compare` can easily be defined using the method `compareTo` as a helping method.

Display 12.8 **Quick Sort Realization of Sorting Pattern *(Part 1 of 2)***

```
/**
 Class that realizes the divide-and-conquer sorting pattern and
 uses the quick sort algorithm.
*/
public class QuickSort
{

    /**
     Precondition: Interval a[begin] through a[end] of a have elements.
     Postcondition: The values in the interval have
     been rearranged so that a[begin] <= a[begin+1] <= ... <= a[end].
    */
    public static void sort(double[] a, int begin, int end)
    {
        if ((end - begin) >= 1)
        {
            int splitPoint = split(a, begin, end);
            sort(a, begin, splitPoint);
            sort(a, splitPoint + 1, end);
            join(a, begin, splitPoint, end);
        }//else sorting one (or fewer) elements so do nothing.
    }

    private static int split(double[] a, int begin, int end)
    {
        double[] temp;
        int size = (end - begin + 1);
        temp = new double[size];

        double splitValue = a[begin];
        int up = 0;
        int down = size - 1;

        //Note that a[begin] = splitValue is skipped.
        for (int i = begin + 1; i <= end; i++)
        {
            if (a[i] <= splitValue)
            {
                temp[up] = a[i];
                up++;
            }
            else
            {
                temp[down] = a[i];
                down--;
            }
        }
```

The method `sort` is identical to the version in the pattern (Display 12.5) except that Type is replaced with `double`.

Display 12.8 Quick Sort Realization of Sorting Pattern *(Part 2 of 2)*

```
        //0 <= up = down < size

        temp[up] = a[begin]; //Positions the split value, spliV.

        //temp[i] <= splitValue for i < up
         // temp[up] = splitValue
         // temp[i] > splitValue for i > up

        for (int i = 0; i < size; i++)
            a[begin + i] = temp[i];

        return (begin + up);
    }

    private static void join(double[] a, int begin,
                                        int splitPoint, int end)
    {
        //Nothing to do.
    }

}
```

EFFICIENCY OF THE SORTING PATTERN ✜

Essentially any sorting algorithm can be realized using this sorting pattern. However, the most efficient implementations are those for which the `split` method divides the array into two substantial size chunks, such as half and half, or one-forth and three-fourths. A realization of `split` that divides the array into one or a very few elements and the rest of the array will not be very efficient.

For example, the merge sort realization of `split` divides the array into two roughly equal parts, and merge sort is indeed very efficient. It can be shown (although we will not do so here) that merge sort has a worst-case running time that is the best possible "up to an order of magnitude."

The quick sort realization of `split` divides the array into two portions that might be almost equal or might be very different in size depending on the choice of a splitting value. Since in extremely unfortunate cases the split might be very uneven for most cases, the worst-case running time for quick sort is not as fast as that of merge sort. However, in practice, quick sort turns out to be a very good sorting algorithm and usually preferable to merge sort.

Section sort, which we discussed in Chapter 5, divides the array into two pieces, one with a single element and one with the rest of the array interval. (See Self-Test Exercise 4.) Because of this uneven division, selection sort has a poor running time, although it does have the virtue of simplicity.

Tip

Pragmatics and Patterns

You should not feel compelled to follow all the fine details of a pattern. Patterns are guides, not requirements. For example, we did the quick sort implementation by exactly following the pattern. We did this to have a clean example. In practice we would have taken some liberties. Notice that, with quick sort, the `join` method does nothing. In practice we would simply eliminate the calls to `join`. These calls incur overhead and accomplish nothing. Other optimizations can also be done once the general pattern of the algorithm is clear.

PATTERN FORMALISM

There is a well-developed body of techniques for using patterns. We will not go into the details here. The UML discussed in Section 10.1 is one formalism used to express patterns. The place within the software design process of patterns and any specific formalisms for patterns is not yet clear. However, it is clear that the basic idea of patterns, as well as certain pattern names, such as *Model-View-Controller*, have become standard and useful tools for software design.

Self-Test Exercises

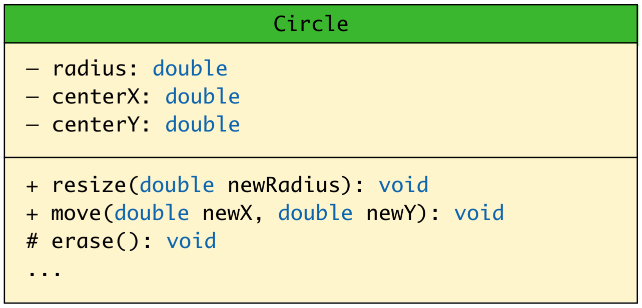
4. Give an implementation of the divide-and-conquer sorting pattern (Display 12.5) that will realize the selection sort algorithm (Display 6.9) for an array with base type `double`.
5. Which of the following would give the fastest run time when an array is sorted using the quick sort algorithm: a fully sorted array, an array of random values, or an array sorted from largest to smallest (that is, sorted backward)? Assume all arrays are of the same size and have the same base type.

Chapter Summary

- The Unified Modeling Language (UML) is a graphical representation language for object-oriented software design.
- Patterns are design principles that apply across a variety of software applications.
- The patterns discussed in this chapter are the Container-Iterator, Adaptor, Model-View-Controller, and Divide-and-Conquer Sorting patterns.
- UML is one formalism that can and is used to express patterns.

Answers to Self-Test Exercises

1. There are many correct answers. Below is one:



```
Circle
– radius: double
– centerX: double
– centerY: double

+ resize(double newRadius): void
+ move(double newX, double newY): void
# erase(): void
...
```

2. The method `sameName` is not listed in the class diagram for `Student`. So, you follow the arrow to the class diagram for `Person`. The method `sameName` with a single parameter of type `Person` is in the class diagram for `Person`. Since you know a `Student` is a `Person`, you know that this definition works for the method `sameName` with a single parameter of type `Student`. So, the definition used for the method `sameName` is in the class definition of `Person`.

3. You start at the class diagram for `Student`. The method `setStudentNumber` with a single parameter of type `int` is in the class diagram for `Student`, so you need look no further. The definition used for the method `setStudentNumber` is in the class definition of `Student`.

4. The code for this is also on the CD that comes with this book. This code is in the file `SelectionSort.java`. A demonstration program is in the file `SelectionSort–Demo.java`.

extra code on CD

```
public class SelectionSort
{
    public static void sort(double[] a,
                                        int begin, int end)
    {
        if ((end – begin) >= 1)
        {
            int splitPoint = split(a, begin, end);
            sort(a, begin, splitPoint);
            sort(a, splitPoint + 1, end);
            join(a, begin, splitPoint, end);
        }//else sorting one (or fewer) elements
         //so do nothing.
    }
    private static int split(double[] a,
                                      int begin, int end)
    {
        int index = indexOfSmallest(begin, a, end);
        interchange(begin,index, a);

         return begin;
    }

    private static void join(double[] a, int begin,
                                    int splitPoint, int end)
    {
      //Nothing to do.
    }

    private static int indexOfSmallest(int startIndex,
                                 double[] a, int endIndex)
    {
        double min = a[startIndex];
        int indexOfMin = startIndex;
        int index;
        for (index = startIndex + 1;
                             index < endIndex; index++)
            if (a[index] < min)
            {
                min = a[index];
                indexOfMin = index;
                //min is smallest of a[startIndex]
                //through a[index]
            }
        return indexOfMin;
    }

    private static void interchange(int i, int j, double[] a)
    {
            double temp;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp; //original value of a[i]
    }
}
```

5. An array of random values would have the fastest run time, since it would divide the array segments into approximately equal subarrays most of the time. The other two cases would give approximately the same running time and would be significantly slower, because the algorithms would always divide an array segment into very unequal size pieces, one piece with only one element and one piece with the rest of the elements. It is ironic but true that our version of the quick sort algorithms has its worst behavior on an already sorted array. There are variations on the quick sort algorithms that perform well on a sorted array. For example, choosing the middle element as the splitting value will give good performance on an already sorted array. But, whatever splitting value you choose, there will always be a few cases with slow running time.

Programming Projects

1. Recode the `QuickSort` class implementation by adding two efficiency improvements to the method `sort`: (1) Eliminate the calls to `join`, since it accomplishes nothing. (2) Add code for the special case of an array of exactly two elements and make the general case apply to arrays of three or more elements.
2. Redo the `QuickSort` class so that it chooses the splitting point as follows: The splitting point is the middle (in size) of the first element, the last element, and an element at approximately the middle of the array. This will make a very uneven split less likely.
3. Redo the `QuickSort` class to have the modifications given for projects 1 and 2.

For additional online Programming Projects, click the CodeMate icons below.

12.4