CASE STUDY Designing a Telephone Directory Program

Problem You have a client who wants to store a simple telephone directory in her computer that she can use for storage and retrieval of names and numbers. She has a data file that contains the names and numbers of her friends. She wants to be able to insert new names and numbers, change the number for an entry, and retrieve selected telephone numbers. She also wants to save any changes in her data file.

Input/Output Requirements

Earlier we discussed some questions that would have to be answered in order to complete the specification of the requirements for the phone directory problem. Most of the questions dealt with input and output considerations. We will list some answers to these questions next.

INPUTS

Initial phone directory	Each name and number will be read from separate lines of a text file. The entries will be read in sequence until all entries are read.
Additional entries	Each entry is typed by the user at the keyboard when requested.

OUTPUTS

Name and phone numbers	The name and number of each person selected by the program user are displayed on separate output lines.
Updated phone directory	Each name and number will be written to separate lines of a text file. The entries will be written in sequence until all entries are written.

Analysis The first step in the analysis is to study the problem input and output requirements carefully to make sure that they are understood and make sense. You can use a tool called a *use case* to help you refine the system requirements.

Use Cases

A use case is a list of the user actions and system responses for a particular subproblem in the order that they are likely to occur.

The following four subproblems were identified for the telephone directory program:

- Read the initial directory from an existing file
- Insert a new entry
- Edit an existing entry
- Retrieve and display an entry

The use case (Table 1.4) for the first subproblem ("Read the initial directory") shows that the user issues a single command and the system responds by either reading a directory from a file or by creating an empty directory if there is no file. The second use case (Table 1.5) is for the subproblems "Insert a new entry" and "Edit an existing entry". Because the names in the directory must be unique, inserting a new entry and editing an existing entry require a search to determine whether the name is already present. Thus, from the user's point of view, the insert and edit processes are the same. The last use case (Table 1.6) shows the user interaction for the last subproblem ("Retrieve and display an entry").

The steps shown in each use case flesh out the user interaction with the program. The use cases should be reviewed by the client to make sure that your intentions are the same as hers. For most of the problems we study in this book, the user interaction is straightforward enough that use cases will not be required.

TABLE 1.4
Use Case for Reading the Initial Directory

Step	User's Action	System's Response
1.	User issues a command to the operating system to load and run the Phone Directory program, specifying the name of the file that contains the directory.	
2.		The Phone Directory program is started, and the directory contents initialized from the data file. If the data file does not exist, an initially empty directory is created.

TABLE 1.5
Use Case for Inserting a New Entry or Editing an Existing Entry

Step	User's Action	System's Response
1.	User issues the command to insert or change an entry.	
2.		System prompts for the name.
3.	User enters name.	If user cancels entry of name, process terminates.
4.		System prompts for the number.
5.	User enters number.	If user cancels entry of number, process terminates.
6.		The directory is updated to contain the new name and number. If the name was not already in the directory, the user is notified that a new name was entered. If the name already exists, the user is notified that the number was changed and is shown both the old and new numbers.

TABLE 1.6
Use Case for Retrieving and Displaying an Entry

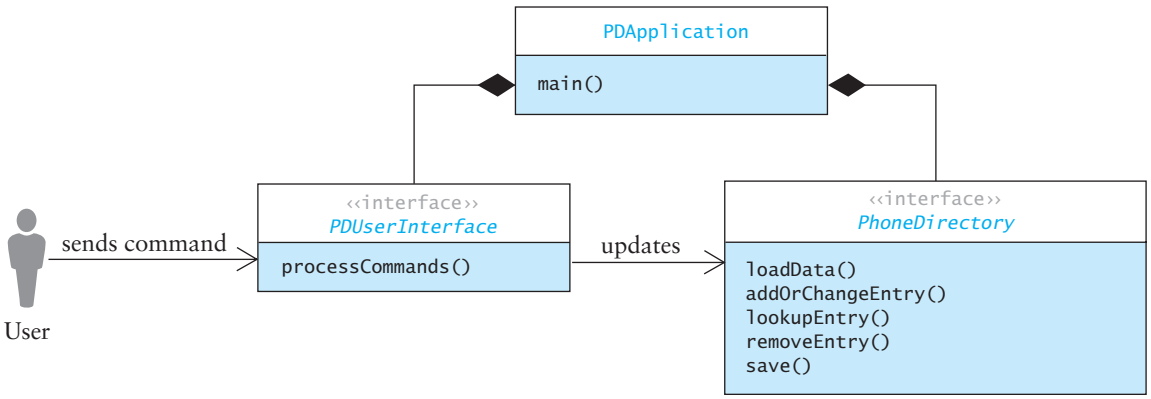
Step	User's Action	System's Response
1.	User issues the command to retrieve and display an entry.	
2.		System prompts for the name.
3.	User enters name.	If user cancels entry of name, process terminates.
4.		The system retrieves the entry from the directory. If found, the name and number are displayed; otherwise, a message is displayed indicating that the name is not in the directory.

Refinement of Initial Class Diagram

Earlier we used the top-down design approach to identify the subproblems to be solved (see Figures 1.3 to Figure 1.5) and came up with the list of level 1 subproblems shown in the previous section. As discussed, you can combine the second and third subproblems ("Insert a new entry", "Edit an existing entry") and add a subproblem to save the directory. The modified list follows:

- Read the initial directory from an existing file
- Insert a new entry or edit an existing entry.
- Retrieve and display an entry.
- Save the modified directory back to the file.

FIGURE 1.9
Phone Directory Application Class Diagram: Revision 1



The directory should be saved whenever the program is exited. The phone directory has limited usefulness if updates to the directory cannot be saved from one run to the next.

There is another way to split this problem into subproblems. Overall, we want a phone directory application that combines a user interface as the front end and a persistent (permanent) storage of data as the back end. Thus, Figure 1.6 can be refined as shown in Figure 1.9. The black diamonds in Figure 1.9 indicate that a `PDApplication` object has objects of type `PDUserInterface` and `PhoneDirectory` as its components, and that they are created by the `PDApplication` object.

In Section 1.4, we identified two abstract data types: the `PDUserInterface` and the `PhoneDirectory`. They are shown in the class diagram as interfaces. In Section 1.6 classes that implement these interfaces will be designed. By splitting the design between the user interface and the directory, we can work on them independently. As long as the requirements defined by the interfaces are met, the front-end user interface does not care which back end it is dealing with, and the back-end directory does not care which front end it is dealing with.

Design Overview of Classes and Their Interaction

Next, we identify all classes that will be part of the problem solution and describe their interaction. Besides the classes that implement the two interfaces shown in Figure 1.9, classes from the Java API will be used to perform input/output. We also need a class with a `main` method. Table 1.7 shows a summary of some of the classes and interfaces that will be used in our solution.

TABLE 1.7
Summary of Classes and Interfaces Used in Phone Directory Solution

Class/Interface	Description
`PDApplication`	Contains the `main` method. It instantiates classes that implement the `PhoneDirectory` and `PDUserInterface` interfaces.
`DirectoryEntry`	Contains a name-number pair.
`PhoneDirectory`	The interface that specifies methods to retrieve, insert, modify, load, and save the phone directory.
`PDUserInterface`	The interface that defines the user interface, which accepts commands from the user and calls the appropriate methods in the `PhoneDirectory` class to perform the desired action.
`BufferedReader`	A class in the Java API that breaks a stream of input characters into lines through a `Reader` object.
`PrintWriter`	A class in the Java API that provides output lines through a `Writer` object.

The first class in Table 1.7, `PDApplication`, contains a `main` method which starts program execution. From the use case in Table 1.4, we know that this method must create the `PhoneDirectory` object and read in the initial directory. Next, it must create a `PDUserInterface` object that interacts with the user to determine which operations should be performed. The list of steps for method `main` follows.

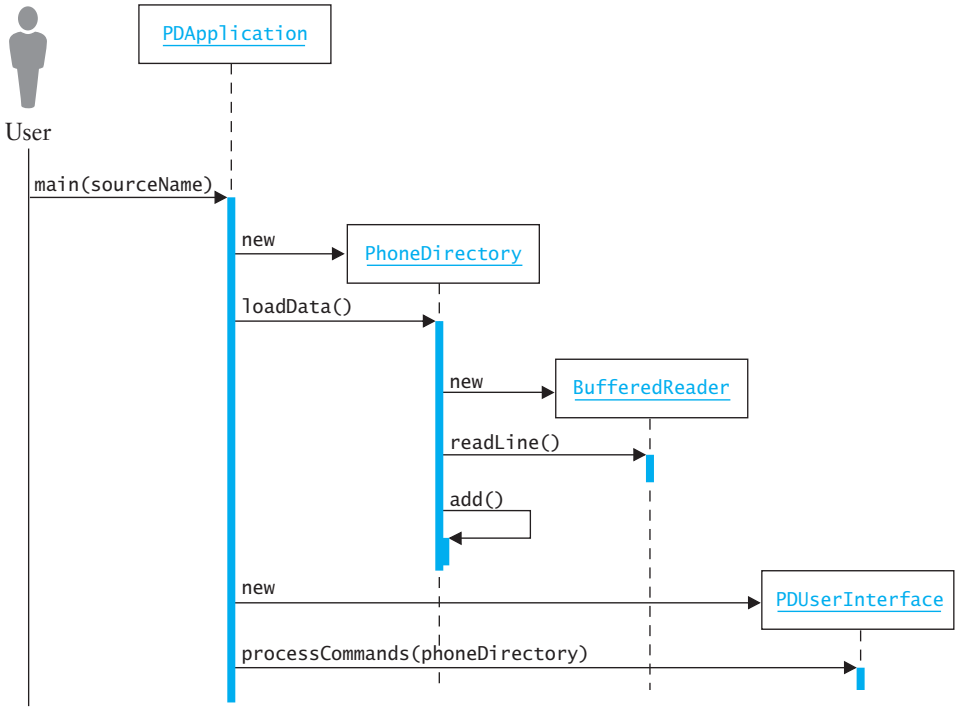
Algorithm for `main` Method

1. Create a new `PhoneDirectory` object.
2. Send it a message to read the initial directory data from a file.
3. Create a new `PDUserInterface` object.
4. Send it a message to perform all user operations on the `PhoneDirectory` object.

To perform Step 4, the `PDUserInterface` method `processCommands` will call its own internal methods that will, in turn, call `PhoneDirectory` methods that perform the specified operation on the `PhoneDirectory` object.

Next we show the UML *sequence diagram* for the `main` method. A sequence diagram (see Appendix B) is an OOD tool that documents the interaction between the objects in a program. Sequence diagrams are used to show the flow of information through the program and to identify the messages that are passed from one object to another.

FIGURE 1.10
Sequence Diagram for `main` Method



Sequence Diagram for `main` Method

The sequence diagram for the `main` method is shown in Figure 1.10. The first (and only) parameter for `main` will be the name of the file containing the directory data. We show this event in the sequence diagram as the user issuing the message `main(sourceName)` to the `PDApplication` class.

The sequence diagram shows all the objects involved in this use case across the horizontal axis, with each object's type underlined. Time is shown along the vertical axis. There is a dashed line coming down from each object that represents the object's *life line*. When a method of this object is called, the dashed line becomes a solid line indicating that the method is executing. All interaction with an object is indicated by a horizontal line that terminates at the object's life line.

The `PDApplication` object is created when the application begins execution. Tracing down its life line, you can see that its `main` method first sends the `new` message to a class that implements the `PhoneDirectory` interface, creating a new `PhoneDirectory` object. Next, `main` sends that object the `loadData` message. Method `loadData` is

described in the PhoneDirectory interface. Looking at the life line for this PhoneDirectory object, you see that method loadData creates a new BufferedReader object and sends it a readLine message. Next, loadData sends the add message to the PhoneDirectory object. (Note that add is a new method that was not identified earlier.) This is the same object as the one that received the loadData message, so this add message is known as a *message to self*. Although the sequence diagram cannot show looping, the process of reading lines and adding entries continues until there are no remaining entries.

After all entries are read and saved, method main creates a new object (type PDUserInterface) and sends it the processCommands message, passing it the PhoneDirectory object as an argument. This provides the PDUserInterface object the necessary access to the PhoneDirectory object to process the commands. This completes the sequence diagram for reading the initial directory from a file. It shows all the steps performed by method main, including calling processCommands after the directory is loaded. Method processCommands will continue executing until the user issues the "Exit program" command.

The sequence diagram (Figure 1.10) shows that method loadData of the PhoneDirectory object performs most of the work for the "Read initial directory data" use case. Method loadData calls all the methods shown after it on the life line for the PhoneDirectory object.

CASE STUDY Designing a Telephone Directory Program (cont.)

Design Design of Data Structures for the Phone Directory

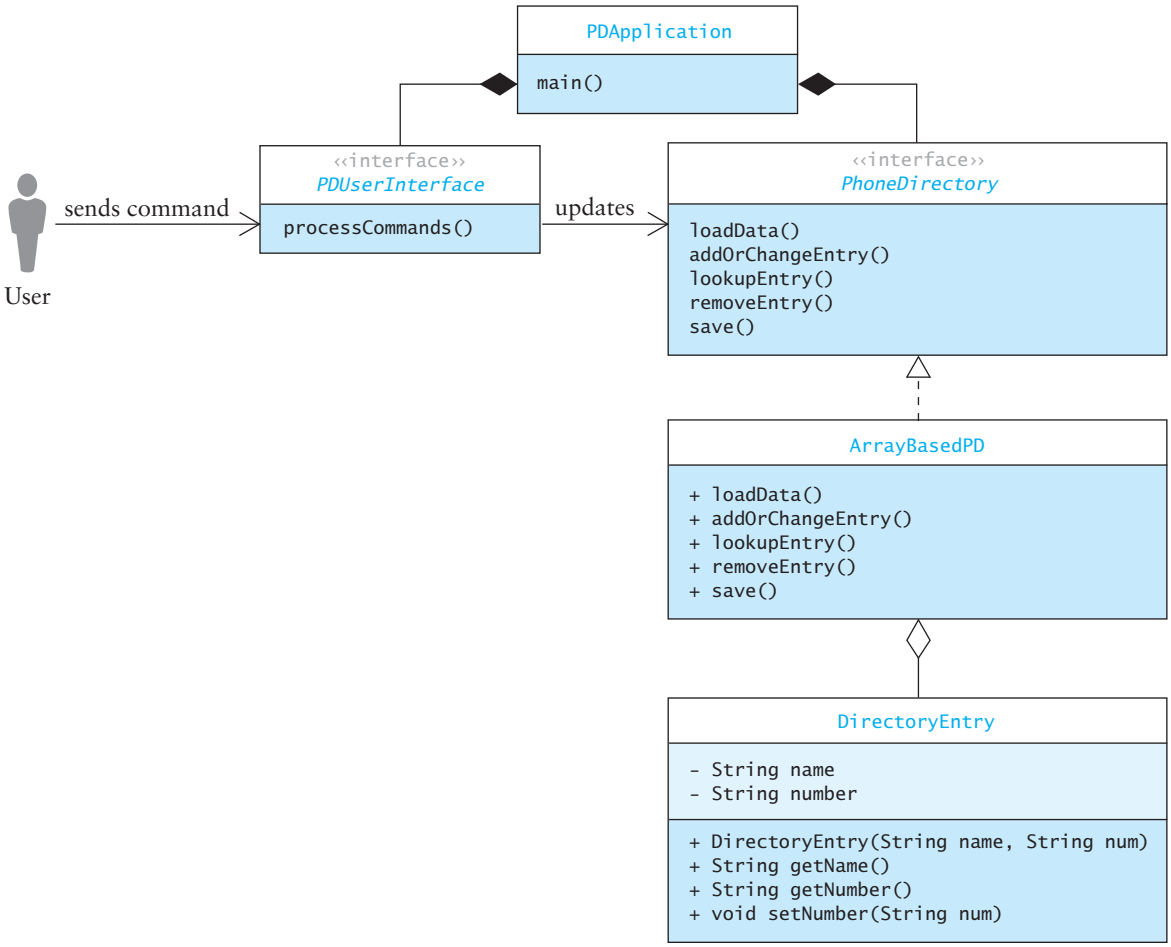
Next, we consider the actual data elements that will be involved in the telephone directory problem. We will define a class `DirectoryEntry`, which will contain the name-number pairs, and a class `ArrayBasedPD`, which implements the `PhoneDirectory` interface. This class will contain an array of `DirectoryEntrys`. In later chapters we will show alternative designs that use classes that are part of the Java API (for example, class `ArrayList`).

Our new class diagram is shown in Figure 1.10. The open diamond indicates that `DirectoryEntry` objects are components of `ArrayBasedPD` objects, but they can also be associated wth other objects (for example, the data file). For class `DirectoryEntry`, we show data fields (attributes) in the light-color screen and methods in the darker-color screen. Next, we discuss the two actual classes shown in this diagram: `DirectoryEntry` and `ArrayBasedPD`.

SYNTAX UML Syntax

In UML class diagrams, the + sign next to the method names indicate that these methods are public. The – sign next to the attributes `name` and `number` indicate that they are private. For the class `DirectoryEntry` we show the types of the attributes, and the parameter types and return types of the methods. Showing this information on the diagram is optional. We will generally show this information in separate tables such as Table 1.7. Appendix B provides a summary of UML.

FIGURE 1.11
Phone Directory Application Class Diagram: Revision 2



Design of the `DirectoryEntry` Class

The `DirectoryEntry` objects will contain the name-and-number pairs. The name is immutable; that is, it cannot be changed. For the purposes of your design, if you need to change the name of a person in your directory, you must remove the old entry and create a new one. The number, however, can be changed. Thus a straightforward design consists of

- Two data fields: `name` and `number`
- A constructor that sets both `name` and `number`

TABLE 1.8
Design of the `DirectoryEntry` Class

Data Field	Attribute
`private String name`	The name of the individual represented in the entry.
`private String number`	The phone number for this individual.
Constructor	**Behavior**
`public DirectoryEntry(String name, String number)`	Creates a new `DirectoryEntry` with the specified `name` and `number`.
Method	**Behavior**
`public String getName()`	Retrieves the `name`.
`public String getNumber()`	Retrieves the `number`.
`public void setNumber(String number)`	Sets the `number` to the specified value.

- Accessor methods for both `name` and `number`
- A mutator method for `number`

This design is shown in Table 1.8.

Design of the `ArrayBasedPD` Class

The `ArrayBasedPD` class implements `PhoneDirectory`. We showed a portion of this interface earlier (Listing 1.1); Table 1.9 shows the methods for the interface and Listing 1.2 shows the complete interface.

TABLE 1.9
Methods Declared in Interface `PhoneDirectory`

Method	Behavior
`public void loadData(String sourceName)`	Loads the data from the data file whose name is given by `sourceName`.
`public String addOrChangeEntry (String name, String number)`	Changes the `number` associated with the given `name` to the new value, or adds a new entry with this `name` and `number`.
`public String lookupEntry (String name)`	Searches the directory for the given `name`.
`public String removeEntry (String name)`	Removes the entry with the specified name from the directory and returns that person's number or **`null`** if not in the directory (left as an exercise).
`public void save()`	Writes the contents of the array of directory entries to the data file.

LISTING 1.2
PhoneDirectory.java

```
/** The interface for the telephone directory.
*/
public interface PhoneDirectory {

    /** Load the data file containing the directory, or
        establish a connection with the data source.
        @param sourceName The name of the file (data source)
                          with the phone directory entries
    */
    void loadData(String sourceName);

    /** Look up an entry.
        @param name The name of the person to look up
        @return The number or null if name is not in the directory
    */
    String lookupEntry(String name);

    /** Add an entry or change an existing entry.
        @param name The name of the person being added or changed
        @param number The new number to be assigned
        @return The old number or, if a new entry, null
    */
    String addOrChangeEntry(String name, String number);

    /** Remove an entry from the directory.
        @param name The name of the person to be removed
        @return The current number. If not in directory, null is
                returned
    */
    String removeEntry(String name);

    /** Method to save the directory.
        pre:  The directory has been loaded with data.
        post: Contents of directory written back to the file in the
              form of name-number pairs on adjacent lines
              modified is reset to false.
    */
    void save();
}
```

Class `ArrayBasedPD` must implement these methods. It must also declare a data field for storage of the phone directory. Table 1.10 describes the data fields of class `ArrayBasedPD`. In addition to the array of directory entries, the class includes data fields to help keep track of the array `size` and `capacity` and whether it has been `modified`. The methods will be designed in the next section.

TABLE 1.10
Data Fields of Class `ArrayBasedPD`

Data Field	Attribute
`private static final int INITIAL_CAPACITY`	The initial capacity of the array to hold the directory entries.
`private int capacity`	The current capacity of the array to hold the directory entries.
`private int size`	The number of directory entries currently stored in the array.
`private DirectoryEntry[] theDirectory`	The array of directory entries.
`private String sourceName`	The name of the data file.
`private boolean modified`	A `boolean` variable to indicate whether the contents of the array have been modified since they were last loaded or saved.

Design of `ArrayBasedPD` Methods

In this section you will complete the design of the `ArrayBasedPD` class. At this stage you need to specify the method algorithms. We will develop *pseudocode* descriptions of the algorithms. Pseudocode is a combination of English and Java language constructs.

Method `loadData`

Method `loadData` is used to read the initial directory from a data file. The file name is passed as an argument to `loadData` when it is called.

Algorithm for Method `loadData`

1. Create a `BufferedReader` for the input file.
2. Read the name.
3. **`while`** the name is not **`null`**
4. Read the number.
5. Add a new entry using method `add`.
6. Read the name.

Note that we have identified a new method, `add`, for class `ArrayBasedPD`.

Method `addOrChangeEntry`

Method `addOrChangeEntry` is used to either add a new entry to the directory or change an existing entry if the name is already in the directory. The name and number are passed as arguments to `addOrChangeEntry`.

Algorithm for Method addOrChangeEntry

1. Call method `find` to see whether the name is in the directory.
2. **`if`** the name is in the directory
3. Change the number using the `setNumber` method of the `DirectoryEntry`.
4. Return the previous value of the number.

   `else`
5. Add a new entry using method `add`.
6. Return **`null`**.

Note that we have identified another new method, `find`, for class `ArrayBasedPD`.

Method `lookupEntry`

Method `lookupEntry` is passed a person's name as an argument. It retrieves the person's number or **`null`** if the name is not found.

Algorithm for `lookupEntry`

1. The `PhoneDirectory` object uses its internal `find` method to locate the entry.
2. **`if`** the entry is found
3. `DirectoryEntry`'s `getNumber` method retrieves the number, which is returned to the caller.

   `else`
4. **`null`** is returned.

Method `save`

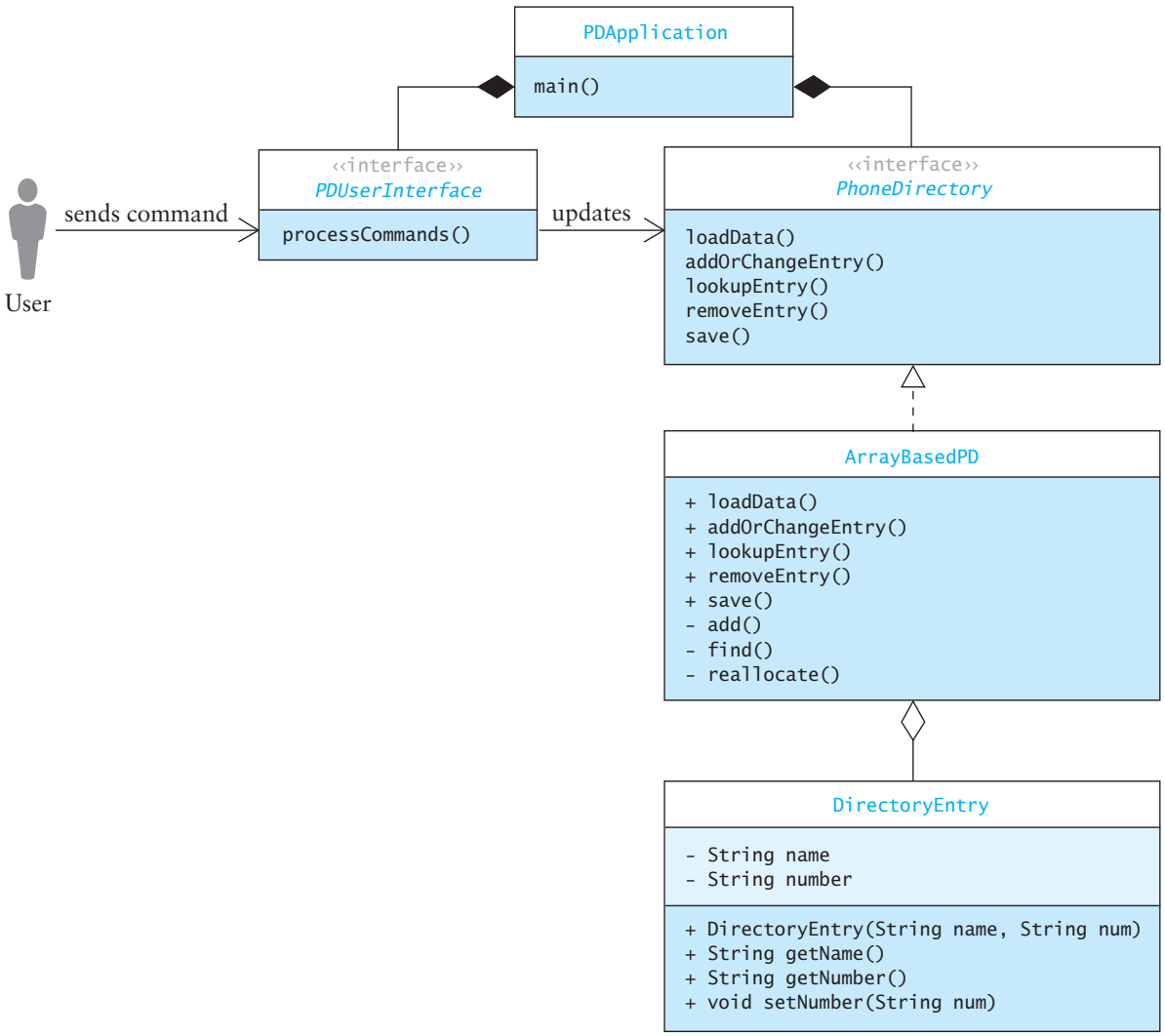
Method `save` creates an output file and then writes all information stored in the array to this file. The file name is stored in data field `sourceName`. The algorithm for the `save` method follows.

Algorithm for save

1. Create a `PrintWriter` object associated with the file.
2. **`for`** each entry in the array
3. Call `getName` to get the name from the entry.
4. Write the name on a line.
5. Call `getNumber` to get the number from the entry.
6. Write the number on a line.
7. Close the `PrintWriter` object.

Figure 1.12 shows the final class diagram with the additional methods.

FIGURE 1.12
Phone Directory Application Class Diagram: Revision 3



CASE STUDY Designing a Telephone Directory Program (cont.)

Implementation

Next we write the code for class `ArrayBasedPD`. Listing 1.3 shows the data field declarations for the class. We use the Javadoc style for commenting the data fields.

LISTING 1.3
Data Field Declarations for `ArrayBasedPD.java`

```
import java.io.*;

/** This is an implementation of the PhoneDirectory interface that uses
    an array to store the data.
*/
public class ArrayBasedPD implements PhoneDirectory {

    // Data Fields

    /** The initial capacity of the array */
    private static final int INITIAL_CAPACITY = 100;

    /** The current capacity of the array */
    private int capacity = INITIAL_CAPACITY;

    /** The current size of the array (number of directory entries) */
    private int size = 0;
```

```
    /** The array to contain the directory data */
    private DirectoryEntry[] theDirectory =
        new DirectoryEntry[capacity];

    /** The data file that contains the directory data */
    private String sourceName = null;

    /** Boolean flag to indicate whether the directory was
        modified since it was either loaded or saved. */
    private boolean modified = false;

    ...
}
```

Coding the Methods

Table 1.11 reviews the private methods of class `ArrayBasedPD`. They are private because they were not declared in the `PhoneDirectory` interface and should not be called by a client. Two of these, `add` and `find`, were discussed previously. Method `reallocate` will be discussed later in this section. Method `removeEntry` is left as an exercise.

The `loadData` Method

The `loadData` method (Listing 1.4) is called by the `main` method of class `PDApplication` to read the initial directory data from an input file (parameter `sourceName`). The data entry process takes place in the **`while`** loop inside the **`try-catch`** statement. (If you are unfamiliar with the use of **`try-catch`** statements for exception handling during file processing, you can review Appendix A. We also discuss these topics in the next chapter.) The **`while`** loop implements Steps 4 through 6 of the algorithm for `loadData` shown earlier. This method reads each name and number from two consecutive data lines and adds that entry to the array.

TABLE 1.11
Private Methods of `ArrayBasedPD` class

Private Method	Behavior
`private int find(String name)`	Searches the array of directory entries for the `name`.
`private void add(String name, String number)`	Adds a new entry with the given `name` and `number` to the array of directory entries.
`private void removeEntry(int index)`	Removes the entry at the given `index` from the directory array.
`private void reallocate()`	Creates a new array of directory entries with twice the capacity of the current one.

LISTING 1.4

Method `loadData` for `ArrayBasedPD.java`

```
/** Method to load the data file.
    pre:  The directory storage has been created and it is empty.
          If the file exists, it consists of name-number pairs
          on adjacent lines.
    post: The data from the file is loaded into the directory.
    @param sourceName The name of the data file
*/
public void loadData(String sourceName) {
    // Remember the source name.
    this.sourceName = sourceName;
    try {
        // Create a BufferedReader for the file.
        BufferedReader in = new BufferedReader(
            new FileReader(sourceName));
        String name;
        String number;

        // Read each name and number and add the entry to the array.
        while ((name = in.readLine()) != null) {
            // Read name and number from successive lines.
            if ((number = in.readLine()) == null) {
                break;     // No number read, exit loop.
            }
            // Add an entry for this name and number.
            add(name, number);
        }

        // Close the file.
        in.close();
    } catch (FileNotFoundException ex) {
        // Do nothing - no data to load.
        return;
    } catch (IOException ex) {
        System.err.println("Load of directory failed.");
        ex.printStackTrace();
        System.exit(1);
    }
}
```

The `readLine` method of the `BufferedReader` class reads a line and returns it as a `String` object. If there is no more data to be read, **`null`** is returned, so we exit the **`while`** loop. Note that we combined the assignment statement and the test for **`null`** in the **`while`** statement condition

```
while ((name = in.readLine()) != null) {...}
```

Similarly we combined the assignment and test when reading the number in the **`if`** condition

```
if ((number = in.readLine()) == null) {
    break;    // No number read, exit loop.
}
```

Therefore, we also exit the loop (through execution of a **break** statement) if a `name` was read but a `number` was not. If both a name and number are read, then a new entry is added to the directory and we continue reading and adding entries.

If a file with name `sourceName` is not found, we immediately return (no data to read). (Later, we will write the new directory to file `sourceName`.) If an input/output error occurs, a stack trace is displayed and we exit the program with an exit code of 1, indicating an error.

PROGRAM STYLE

Use of Assignment in a Condition and Use of break

The **while** loop in this section uses two features of Java that simplify the code but are considered controversial. Some programmers prefer not to combine assignment with the evaluation of a condition. However, in this case, it simplifies the code to do it this way. Also, the requirement to exit the loop without storing an entry is met very naturally using the **break** statement. Some programmers prefer to provide only one way to exit a loop: when the **while** condition fails.

The `addOrChangeEntry` Method

This method calls the internal method `find` to locate the name in the array. Method `find` will either return the index of the entry, or return –1 (minus 1) to indicate that the entry is not in the array. If the entry is in the array, that entry's `setNumber` method is called to change the number; otherwise a new entry is added by calling the `add` method:

```
/** Add an entry or change an existing entry.
    @param name The name of the person being added or changed
    @param number The new number to be assigned
    @return The old number or, if a new entry, null
*/
public String addOrChangeEntry(String name, String number) {
    String oldNumber = null;
    int index = find(name);
    if (index > -1) {
        oldNumber = theDirectory[index].getNumber();
        theDirectory[index].setNumber(number);
    } else {
        add(name, number);
    }
    modified = true;
    return oldNumber;
}
```

The `lookupEntry` Method

This method also uses the internal `find` method to locate the entry in the array. If the entry is located, it is returned; otherwise **`null`** is returned.

```
/** Look up an entry.
    @param name The name of the person
    @return The number. If not in the directory, null is returned
*/
public String lookupEntry(String name) {
    int index = find(name);
    if (index > -1) {
        return theDirectory[index].getNumber();
    } else {
        return null;
    }
}
```

The `save` Method

If the directory has not been modified, method `save` (Listing 1.5) does nothing. Otherwise it creates a `PrintWriter` object and writes all phone directory entries to it. The output file name is the same as the input file name (`sourceName`), so an existing directory file will be overwritten. The **`while`** loop inside the **`try-catch`** statement writes the entries.

LISTING 1.5
Method `save` for `ArrayBasedPD.java`

```
/** Method to save the directory.
    pre:  The directory has been loaded with data.
    post: Contents of directory written back to the file in the
          form of name-number pairs on adjacent lines.
          modified is reset to false.
*/
public void save() {
    if (modified) { // If not modified, do nothing.
        try {
            // Create PrintWriter for the file.
            PrintWriter out = new PrintWriter(
                new FileWriter(sourceName));

            // Write each directory entry to the file.
            for (int i = 0; i < size; i++) {
                // Write the name.
                out.println(theDirectory[i].getName());
                // Write the number.
                out.println(theDirectory[i].getNumber());
            }
```

```
            // Close the file.
            out.close();
            modified = false;
        } catch (Exception ex) {
            System.err.println("Save of directory failed");
            ex.printStackTrace();
            System.exit(1);
        }
    }
}
```

The `find` Method

The `find` method uses a **`for`** loop to search the array for the requested name. If located, its index is returned; otherwise –1 (minus 1) is returned.

```
/** Find an entry in the directory.
    @param name The name to be found
    @return The index of the entry with the requested name.
            If the name is not in the directory, returns -1
*/
private int find(String name) {
    for (int i = 0; i < size; i++) {
        if (theDirectory[i].getName().equals(name)) {
            return i;
        }
    }
    return -1;      // Name not found.
}
```

PITFALL

Returning –1 (Failure) Before Examining All Array Elements

A common logic error is to code the search loop for method `find` as follows:

```
for (int i = 0; i < size; i++) {
    if (theDirectory[i].getName().equals(name)) {
        return i;
    } else {
        return -1;  // Incorrect! - tests only one element.
    }
}
```

This loop incorrectly returns a result after testing just the first element.

The `add` Method

The `add` method checks to see whether there is room in the array by comparing the `size` to the `capacity`. If the `size` is less than the `capacity`, the new entry is stored at the end of the array and `size` is incremented by one after the entry is stored (`size++`). If the `size` is greater than or equal to the `capacity`, then the `reallocate` method is called to increase the size of the array before the new item is inserted.

```
/** Add an entry to the directory.
    @param name The name of the new person
    @param number The number of the new person
*/
private void add(String name, String number) {
    if (size >= capacity) {
        reallocate();
    }
    theDirectory[size] = new DirectoryEntry(name, number);
    size++;
}
```

The `reallocate` Method

This method allocates a new array whose size is twice the current array. Method `System.arraycopy` (see Appendix A) copies the contents of the old array (`theDirectory`) to the new array (`newDirectory`):

```
System.arraycopy(theDirectory, 0, newDirectory, 0, theDirectory.length);
```

and `theDirectory` is changed to refer to the new array. The storage allocated to the old array will be recycled by the Java Virtual Machine (JVM)'s garbage collector.

By doubling the size each time that a reallocation is necessary, we reduce the number of times we need to do this. Surprisingly, if we do this only fourteen times, we can store over 1 million entries.

```
/** Allocate a new array to hold the directory. */
private void reallocate() {
    capacity *= 2;
    DirectoryEntry[] newDirectory = new DirectoryEntry[capacity];
    System.arraycopy(theDirectory, 0, newDirectory, 0,
                     theDirectory.length);
    theDirectory = newDirectory;
}
```

Using a Storage Structure Without Reallocation

In Chapter 4, you will study the `ArrayList` data structure, which will enable you to store a directory of increasing size without needing to reallocate storage. You will see that we can change to a different data structure for storing the directory with very little effort, because the problem solution has been so carefully designed. We will only need to code the methods declared in the `PhoneDirectory` interface so that they perform the same operations on an `ArrayList`.

Testing Class `ArrayBasedPD`

To test this class, you should run it with data files that are empty or that contain a single name-and-number pair. You should also run it with data files that contain an odd number of lines (ending with a name but no number). You should see what happens when the array is completely filled and you try to add a new entry. Does method `reallocate` properly double the array's size? When you do a retrieval or an edit operation, make sure you try to retrieve names that are not in the directory as well as names that are in the directory. If an entry has been changed, verify that the new number is retrieved. Finally, check that all new and edited entries are written correctly to the output file. We will discuss testing further in the next chapter.

CASE STUDY Designing a Telephone Directory Program (cont.)

Analysis Through the description of the interface, we know that a class that implements `PDUserInterface` must contain a public method, `processCommands`, declared as follows in the interface:

```
void processCommands(PhoneDirectory theDirectory);
```

The interface enables clients to use method `processCommands` without knowing the details of its implementation (information hiding). We will introduce new private methods that are called by `processCommands` to perform its tasks, but are unavailable to a client.

The kind of user interaction that will take place will be determined by the input/output facilities used in a class that implements the interface. Three options would be console input, GUI input using a specially designed GUI for this problem, and GUI input using `JOptionPane` dialog windows. We will write classes that use the first and last options. (If you are unfamiliar with any of these Java input/output features, review Appendix A.)

For both classes, method `processCommands` should present a menu of choices to the user:

- Add or Change an Entry
- Look Up an Entry
- Remove an Entry
- Save the Directory Data
- Exit the Program

Design For both classes, method `processCommands` will use a "menu-driven" loop to control the interaction with the user. In a true GUI, a loop would not be necessary. After each command is processed, the menu of choices is displayed again. This process continues until the user selects "Exit the program".

```
do {
    // Get the action to perform from the user.
    // The user's choice will be a number from 0 through 4.
    . . .
    switch (choice) {
        case 0: doAddChangeEntry(); break;
        case 1: doLookupEntry(); break;
        case 2: doRemoveEntry(); break;
        case 3: doSave(); break;
        case 4: doSave(); break;
    }
} while (choice < commands.length - 1);
```

The method `processCommands` calls a private method shown in the foregoing **`switch`** statement to perform the user's choice. Note that method `doSave` is called by the last two cases. We discuss the design and coding of these methods next.

Implementation

The PDGUI Class

Our first class will interact with the user through a GUI. It uses method `showOptionDialog` of the `JOptionPane` class, which is part of the Java Swing API, to present the menu to the user, request data from the user, and display the results to the user. The initial menu is as follows:

Method doAddChangeEntry

The `doAddChangeEntry` method uses the `JOptionPane.showInputDialog` method to request the name and new number. Here are examples of these dialogs:

If the user selects Cancel, a null string is returned. In that case the method will return immediately without changing the directory.

The `PhoneDirectory.addOrChangeEntry` method is called if values are entered for the name and number. A return value of **`null`** indicates that this is a new entry, and a confirmation dialog is displayed by `JOptionPane.showMessageDialog`:

If the name was already in the directory, the previous value of the number is returned, and the confirmation shows both the old and the new number, as follows:

Algorithm for Method doAddChangeEntry

1. Read the name of the entry.
2. Read the number of the entry.
3. Send the `addOrChangeEntry` message to the `PhoneDirectory` object.
4. **`if`** the result of `addOrChangeEntry` was **`null`**
5. The message "*name* was added to the directory" is displayed and the new number is displayed.

   `else`
6. The message "number for *name* was changed", and the old and new number are displayed.

Method doLookupEntry

The `doLookupEntry` method uses the same dialog as `doAddChangeEntry` to request the name. If the user cancels the dialog, the method returns. Otherwise the number is looked up by calling the `PhoneDirectory.lookupEntry` method, and the result is displayed. If the name is not in the directory, a message is displayed. Examples of both are as follows:

Algorithm for doLookupEntry

1. Read the name.
2. Issue a `lookupEntry` message to the `PhoneDirectory` object.
3. **`if`** the result is not **`null`**
4. The name and number are displayed.

   `else`
5. A message indicating that the name is not in the directory is displayed.

Method doSave

The `doSave` method calls the `save` method of the `PhoneDirectory` object.

Listing 1.6 shows the code for the `PDGUI` class. It implements all the methods discussed above.

LISTING 1.6

PDGUI.java

```
import javax.swing.*;

/** This class is an implementation of PDUserInterface
    that uses JOptionPane to display the menu of command choices.
*/
public class PDGUI implements PDUserInterface {

    /** A reference to the PhoneDirectory object to be processed.
        Globally available to the command-processing methods.
    */
    private PhoneDirectory theDirectory = null;

    // Methods
    /** Method to display the command choices and process user
        commands.
        pre:  The directory exists and has been loaded with data.
        post: The directory is updated based on user commands.
        @param thePhoneDirectory A reference to the PhoneDirectory
               to be processed.
    */
    public void processCommands(PhoneDirectory thePhoneDirectory) {

        String[] commands = {"Add/Change Entry",
                             "Look Up Entry",
                             "Remove Entry",
                             "Save Directory",
                             "Exit"};

        theDirectory = thePhoneDirectory;
        int choice;

        do {
            choice = JOptionPane.showOptionDialog(
                null,                                // No parent
                "Select a Command",                  // Prompt message
                "PhoneDirectory",                    // Window title
                JOptionPane.YES_NO_CANCEL_OPTION,    // Option type
                JOptionPane.QUESTION_MESSAGE,        // Message type
                null,                                // Icon
                commands,                            // List of commands
                commands[commands.length - 1]);      // Default choice
            switch (choice) {
                case 0: doAddChangeEntry(); break;
                case 1: doLookupEntry(); break;
                case 2: doRemoveEntry(); break;
                case 3: doSave(); break;
                case 4: doSave(); break;
            }
        } while (choice < commands.length - 1);
        System.exit(0);
    }
```

```
/** Method to add or change an entry.
    pre:  The directory exists and has been loaded with data.
    post: A new name is added, or the value for the name is
          changed, modified is set to true.
*/
private void doAddChangeEntry() {
    // Request the name
    String newName = JOptionPane.showInputDialog("Enter name");
    if (newName == null) {
        return; // Dialog was cancelled.
    }
    // Request the number
    String newNumber = JOptionPane.showInputDialog("Enter number");
    if (newNumber == null) {
        return; // Dialog was cancelled.
    }
    // Insert/change name-number
    String oldNumber = theDirectory.addOrChangeEntry(newName,
                                                     newNumber);
    String message = null;
    if (oldNumber == null) {   // New entry.
        message = newName + " was added to the directory"
                  + "\nNew number: " + newNumber;
    } else {   // Changed entry.
        message = "Number for " + newName + " was changed "
                  + "\nOld number: " + oldNumber
                  + "\nNew number: " + newNumber;
    }
    // Display confirmation message.
    JOptionPane.showMessageDialog(null, message);
}

/** Method to look up an entry.
    pre:  The directory has been loaded with data.
    post: No changes made to the directory.
*/
private void doLookupEntry() {
    // Request the name.
    String theName = JOptionPane.showInputDialog("Enter name");
    if (theName == null) {
        return; // Dialog was cancelled.
    }
    // Look up the name.
    String theNumber = theDirectory.lookupEntry(theName);
    String message = null;
    if (theNumber != null) {   // Name was found.
        message = "The number for " + theName + " is " + theNumber;
    } else {   // Name was not found.
        message = theName + " is not listed in the directory";
    }
    // Display the result.
    JOptionPane.showMessageDialog(null, message);
}
```

```
    /** Method to remove an entry.
        pre:  The directory has been loaded with data.
        post: The requested name is removed, modified is set to true.
    */
    private void doRemoveEntry() {
        // Programming Exercise
    }

    /** Method to save the directory to the data file.
        pre:  The directory has been loaded with data.
        post: The current contents of the directory have been saved
              to the data file.
    */
    private void doSave() {
        theDirectory.save();
    }
}
```

Testing Testing Class PDGUI

To test this class you need a `main` method that creates an `ArrayBasedPD` object and a `PDGUI` object. Method `main` must invoke method `loadData` of class `ArrayBasedPD` to read the directory from a file, passing the file name as an argument; `main` must then invoke method `processCommands` of class `PDGUI`, passing the `PhoneDirectory` object as an argument.

```
/** Program to display and modify a simple phone directory. */
public class PDApplication {

    public static void main (String args[]) {
        // Check to see that there is a command line argument.
        if (args.length == 0) {
            System.err.println("You must provide the name of the file"
                               + " that contains the phone directory.");
            System.exit(1);
        }

        // Create a PhoneDirectory object.
        PhoneDirectory phoneDirectory = new ArrayBasedPD();
        // Load the phone directory from the file.
        phoneDirectory.loadData(args[0]);

        // Create a PDUserInterface object.
        PDUserInterface phoneDirectoryInterface = new PDGUI();
        // Process user commands.
        phoneDirectoryInterface.processCommands(phoneDirectory);
    }
}
```

When this method runs, make sure you test all possible commands. Try exiting without first saving the directory and verify that the file is correctly updated and saved.

Implementation The PDConsoleUI Class

Listing 1.7 shows the code for the `PDConsoleUI` class. This class uses `System.out` to display the menu of choices and results. It also uses `System.in` to read data from the user.

The constructor creates a `BufferedReader` object that is attached to `System.in`. This object has a `readLine` method, which reads a line of input and returns it as a `String`.

Method processCommands

The `readLine` method may throw an `IOException`, so the `processCommands` method uses a **`try-catch`** block to enclose the processing of user commands. Each of the individual methods that process the commands is declared to throw an `IOException`. Thus they do not need a **`try-catch`** block. (We provide thorough coverage of exceptions in Chapter 2.)

Should an `IOException` be thrown, an error message will be written to `System.err` and the program will exit with an exit code of 1 (error).

The initial menu is as follows:

Method doAddChangeEntry

The `doAddChangeEntry` method requests the name followed by the number. The `PhoneDirectory.addOrChangeEntry` method is called after values are entered for the name and number. A return value of **`null`** indicates that this is a new entry, and a confirmation dialog is displayed as follows:

If the name was already in the directory, the previous value of the number is returned, and the confirmation shows both the old and new number as follows:

Method doLookupEntry

The `doLookupEntry` method uses the same prompt as `doAddChangeEntry` to request the name. If the user cancels the data entry, the method returns. Otherwise the number is looked up by calling the `PhoneDirectory.lookupEntry` method, and the result is displayed. If the name is not in the directory, a message is displayed. Examples of both cases follow:

Method doSave

The doSave method calls the save method of the PhoneDirectory.

LISTING 1.7
PDConsoleUI.java

```
import java.io.*;

/** This class is an implementation of PDUserInterface
    that uses the console to display the menu of command choices.
*/
public class PDConsoleUI implements PDUserInterface {

    /** A reference to the PhoneDirectory object to be processed.
        Globally available to the command-processing methods.
    */
    private PhoneDirectory theDirectory = null;
    /** Buffered reader to read from the input console. */
    private BufferedReader in = null;

    // Constructor
    /** Default constructor. */
    public PDConsoleUI() {
        in = new BufferedReader(new InputStreamReader(System.in));
    }

    // Methods
    /** Method to display the command choices and process user
        commands.
        pre:  The directory exists and has been loaded with data.
        post: The directory is updated based on user commands.
        @param thePhoneDirectory A reference to the PhoneDirectory
               to be processed
    */
    public void processCommands(PhoneDirectory thePhoneDirectory) {
        String[] commands = {"Add/Change Entry",
                             "Look Up Entry",
                             "Remove Entry",
                             "Save Directory",
                             "Exit"};

        theDirectory = thePhoneDirectory;
        int choice;
        try {
            do {
                for (int i = 0; i < commands.length; i++) {
                    System.out.println("Select " + i + ": "
                                       + commands[i]);
                }
```

```
                String line = in.readLine();
                if (line != null)
                    choice = Integer.parseInt(line);
                else
                    choice = commands.length - 1;
                switch (choice) {
                    case 0: doAddChangeEntry(); break;
                    case 1: doLookupEntry(); break;
                    case 2: doRemoveEntry(); break;
                    case 3: doSave(); break;
                    case 4: doSave(); break;
                }
            } while (choice < commands.length - 1);
            System.exit(0);
        } catch (IOException ex) {
            System.err.println
                ("IO Exception while reading from System.in");
            System.exit(1);
        }
    }

    /** Method to add or change an entry.
        pre:  The directory exists and has been loaded with data.
        post: A new name is added, or the value for the name is
              changed, modified is set to true.
        @throws IOException - if an IO error occurs
    */
    private void doAddChangeEntry() throws IOException {
        // Request the name.
        System.out.println("Enter name");
        String newName = null;
        newName = in.readLine();
        if (newName == null) {
            return;
        }
        // Request the number.
        System.out.println("Enter number");
        String newNumber = null;
        newNumber = in.readLine();
        if (newNumber == null) {
            return;
        }
        // Insert/change name-number.
        String oldNumber =
                (theDirectory.addOrChangeEntry(newName, newNumber);
        String message = null;
        if (oldNumber == null) {   // New entry.
            message = newName + " was added to the directory"
                      + "\nNew number: " + newNumber;
```

```
            } else {   // Changed entry.
                message = "Number for " + newName + " was changed"
                          + "\nOld number: " + oldNumber
                          + "\nNew number: " + newNumber;
            }
            // Display confirmation message.
            System.out.println(message);
        }

        /** Method to look up an entry.
            pre:  The directory has been loaded with data.
            post: No changes made to the directory.
            @throws IOException - If an IO error occurs
        */
        private void doLookupEntry() throws IOException {
            // Request the name.
            System.out.println("Enter name");
            String theName = null;
            theName = in.readLine();
            if (theName == null) {
                return;   // Dialog was cancelled.
            }
            // Look up the name.
            String theNumber = theDirectory.lookupEntry(theName);
            String message = null;
            if (theNumber != null) {   // Name was found.
                message = "The number for " + theName + " is " + theNumber;
            } else {   // Name was not found.
                message = theName + " is not listed in the directory";
            }
            // Display the result.
            System.out.println(message);
        }

        /** Method to remove an entry.
            pre:  The directory has been loaded with data.
            post: The requested name is removed, modifed is set to true.
            @throws IOException - If there is an IO Error
        */
        private void doRemoveEntry() throws IOException {
            // Programming Exercise
        }

        /** Method to save the directory to the data file.
            pre:  The directory has been loaded with data.
            post: The current contents of the directory have been saved
                  to the data file.
        */
        private void doSave() {
            theDirectory.save();
        }
    }
```