Cmp Sci 187: Introduction to Software Design

Following Chapter 1 of text

(Koffmann and Wolfgang)

Outline

- The software challenge and the software life cycle
- Activities of each phase of the software life cycle
- Using top-down design and object-oriented design
- Managing complexity:
  - Data abstraction
  - Procedural abstraction
  - Information hiding
- Class diagrams document interactions between classes

Outline (2)

- Abstract data types:
  - Role in modeling
  - Implementing them with classes and interfaces
- Use cases: tool to document interaction with a user
- Software design process example:
  - Design and implementation of an array-based telephone directory
- Sequence diagrams: tool for documenting the interaction between multiple classes used in a program

The Software Challenge

- Software is ...
  - Used for a long time
  - Updated and maintained
  - By people who did not write it
- Initial specification may be incomplete
- Specification clarified through extensive interaction between user(s) and system analyst(s)
- Requirements specification needed at the beginning of any software project
- Designers and users should both approve it!

Things Change!

- Users’ needs and expectations change
    - Use reveals limitations and flaws
    - Desire for increased convenience, functionality
    - Desire for increased performance
- Environment changes
    - Hardware, OS, software packages (“software rot”)
    - Need to interact with clients, parent org., etc.
    - Law and regulations change
    - Ways of doing business
    - Style, “cool” factor

The Software Life Cycle

- Software goes through <u>stages</u> as it moves from initial concept to finished product
- The sequence of stages is called a <u>life cycle</u>
- Must design and document software:
    - In an organized way for:
        - Understanding and ...
        - Maintenance (*<u>change</u>*) *after the initial release*
- The maintainer is not necessarily the author!
    - ... and even authors *forget*
    - ... and *no one* can keep all details in mind at once

Software Life Cycle Models: The Waterfall Model

- Simplest way to organizing activities in stages
- Activities are:
  - Performed in sequence
  - Result of one flows (falls) into the next
- The Waterfall Model is simple ... but *unworkable*
  - Fundamental flaw: Assumption that each stage can and must be completed before the next one occurs
  - Example: User may need to see finished product to express true requirements!

Waterfall Model

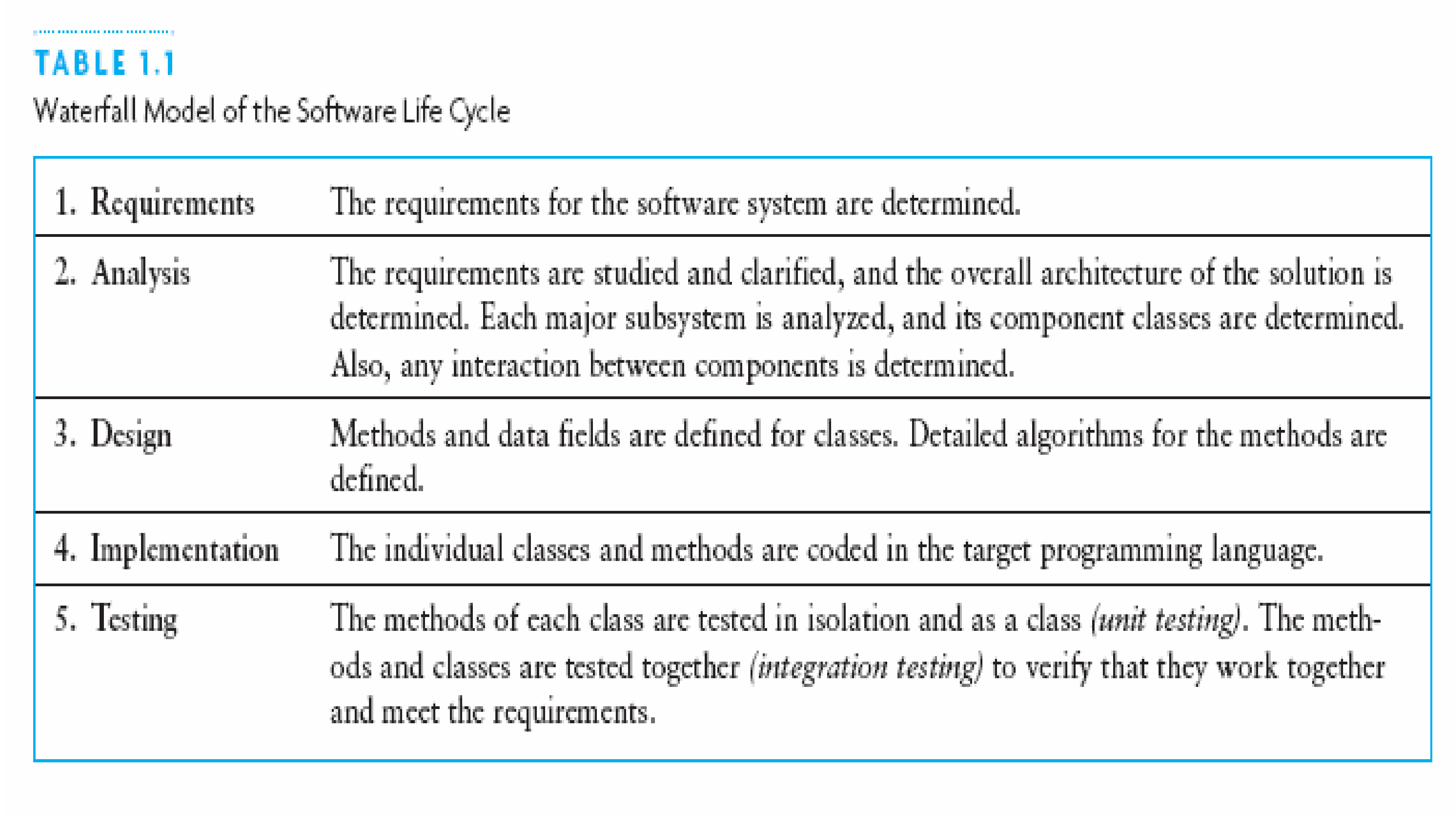
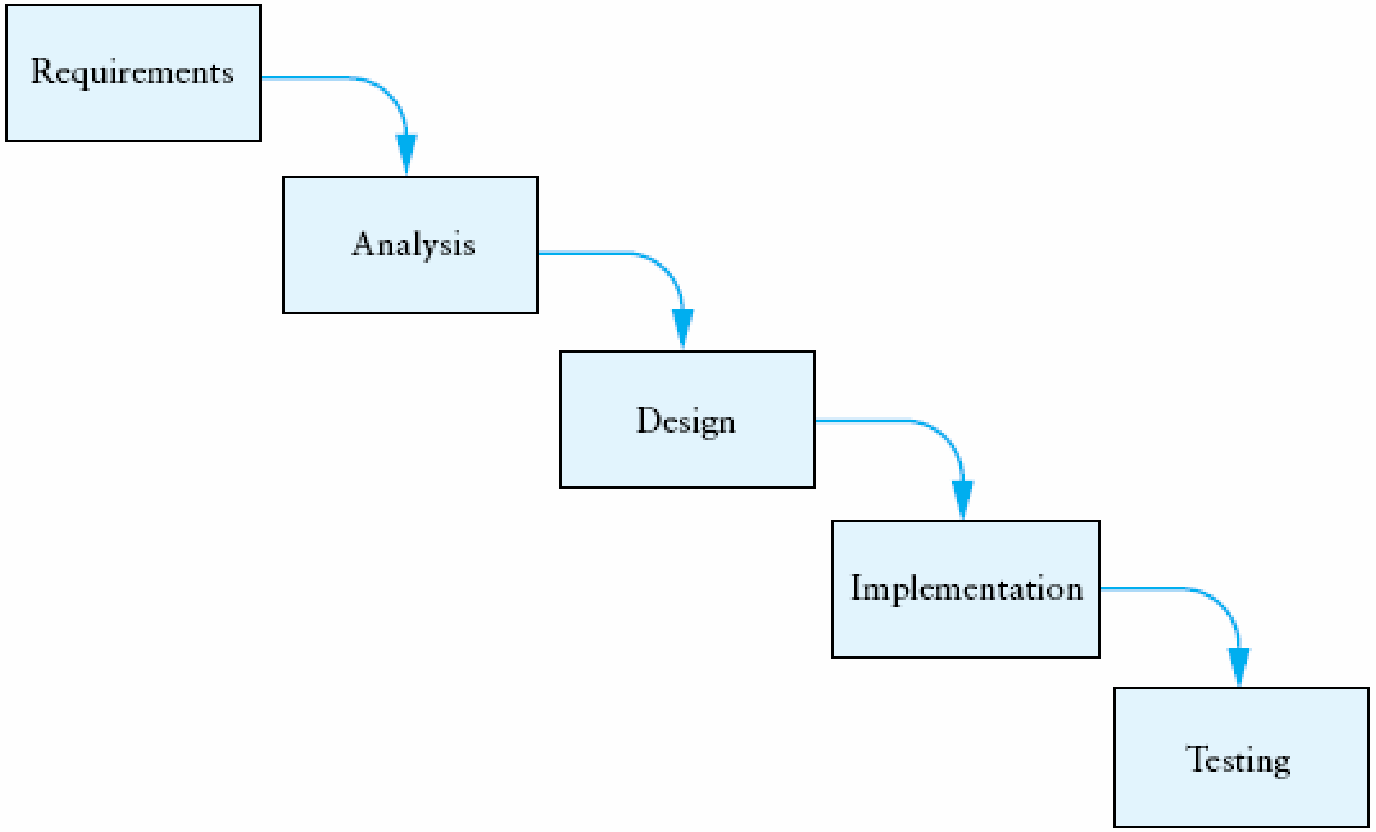


TABLE 1.1
Waterfall Model of the Software Life Cycle

1. Requirements	The requirements for the software system are determined.
2. Analysis	The requirements are studied and clarified, and the overall architecture of the solution is determined. Each major subsystem is analyzed, and its component classes are determined. Also, any interaction between components is determined.
3. Design	Methods and data fields are defined for classes. Detailed algorithms for the methods are defined.
4. Implementation	The individual classes and methods are coded in the target programming language.
5. Testing	The methods of each class are tested in isolation and as a class *(unit testing)*. The methods and classes are tested together *(integration testing)* to verify that they work together and meet the requirements.

Waterfall Model (2)

FIGURE 1.1
The Waterfall Software Life Cycle Model

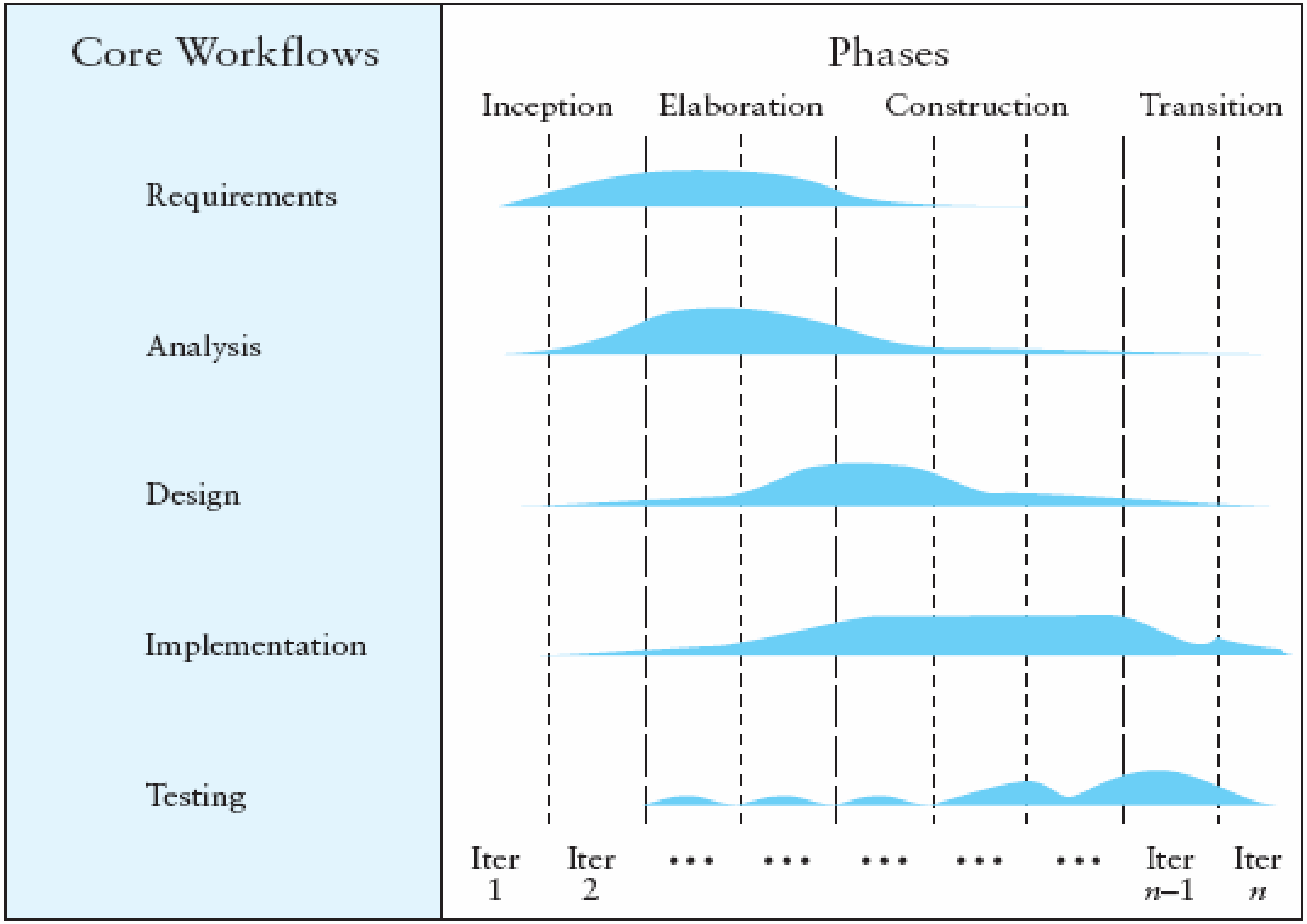


Other Software Life Cycle Models

- Common theme among models: *stages* or *cycles*
- Unified Model:
  - Cycles are called phases and iterations
  - Activities are called workflows
- The four phases of the Unified Model:
  - Inception
  - Elaboration
  - Construction
  - Transition

Other Software Life Cycle Models (2)

FIGURE 1.2
The Unified Software Life Cycle Model



Source: Grady Booch, Ivar Jacobson, and James Rumbaugh, *The Unified Software Development Process* (Addison-Wesley, 1999)

Software Life Cycle Activities

Activities essential for successful development:

- Requirements specification
- Architectural, component, & detailed designs
- Implementation
- Unit, integration, and acceptance testing
- Installation and maintenance

Software Life Cycle Activities Defined

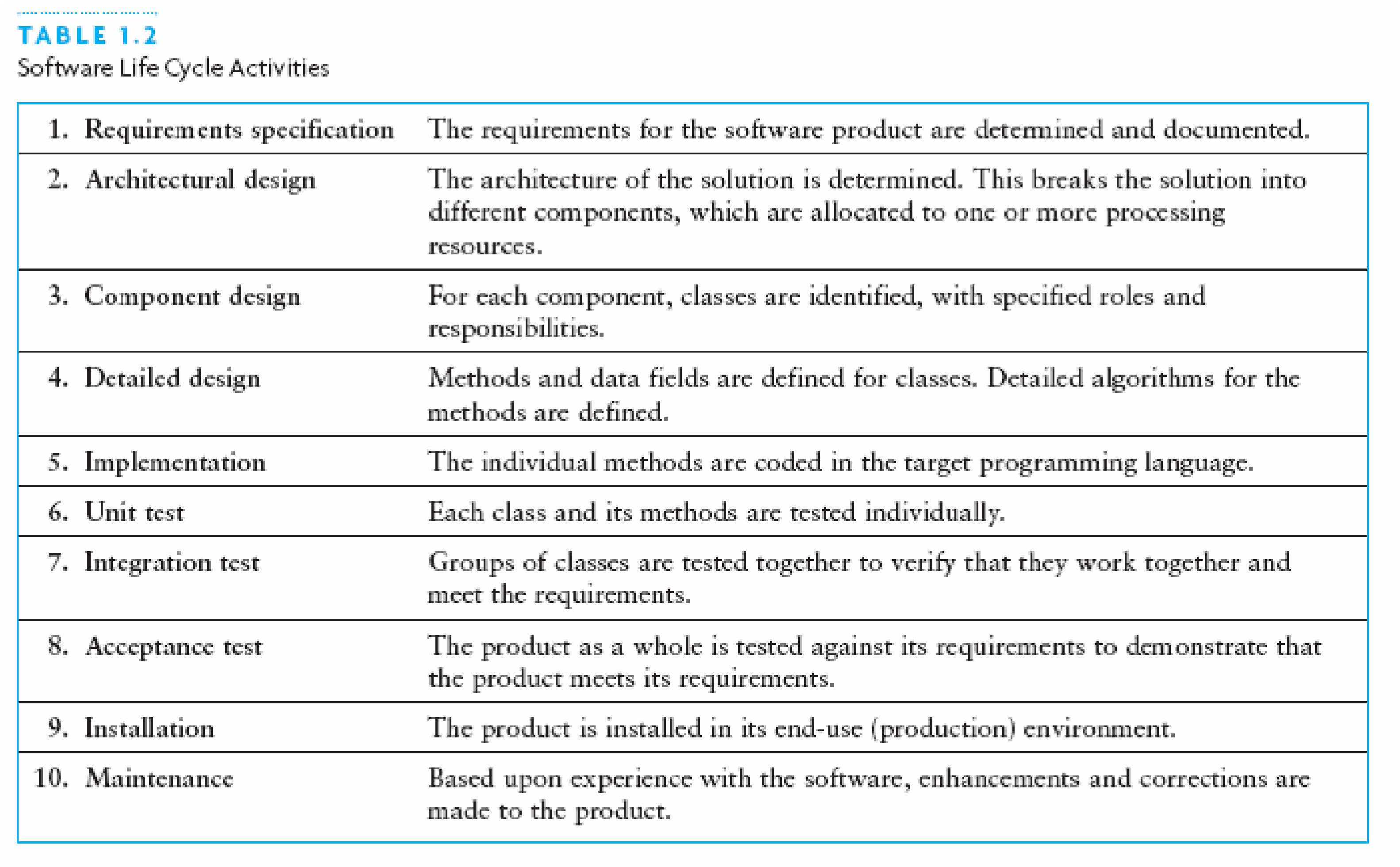


TABLE 1.2
Software Life Cycle Activities

1. Requirements specification	The requirements for the software product are determined and documented.
2. Architectural design	The architecture of the solution is determined. This breaks the solution into different components, which are allocated to one or more processing resources.
3. Component design	For each component, classes are identified, with specified roles and responsibilities.
4. Detailed design	Methods and data fields are defined for classes. Detailed algorithms for the methods are defined.
5. Implementation	The individual methods are coded in the target programming language.
6. Unit test	Each class and its methods are tested individually.
7. Integration test	Groups of classes are tested together to verify that they work together and meet the requirements.
8. Acceptance test	The product as a whole is tested against its requirements to demonstrate that the product meets its requirements.
9. Installation	The product is installed in its end-use (production) environment.
10. Maintenance	Based upon experience with the software, enhancements and corrections are made to the product.

Software Life Cycle Activities (more)

- Requirements Specification
  - System analyst works with users to clarify the detailed system requirements
  - Questions include format of input data, desired form of any output screens, and data validation
- Analysis
  - Make sure you completely understand the problem before starting the design or program a solution
  - Evaluate different approaches to the design

Software Life Cycle Activities (continued)

- Design
  - Top-down: break system into smaller subsystems
  - Object-oriented: identify objects and their interactions
  - UML diagrams: tool to show interactions between:
    - Classes (inside the system)
    - Classes and external entities

Example of Top-Down: Stepwise Refinement

FIGURE 1.3
Structure Chart for Telephone Directory Problem

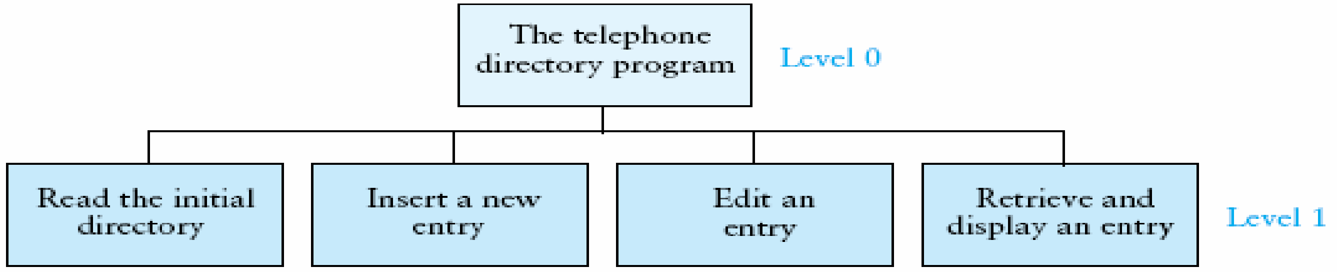

The telephone directory program
Level 0
Read the initial directory
Insert a new entry
Edit an entry
Retrieve and display an entry
Level 1

FIGURE 1.4
Refinement of "Read the initial directory"

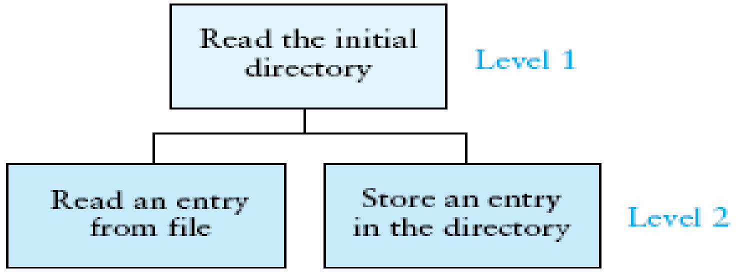
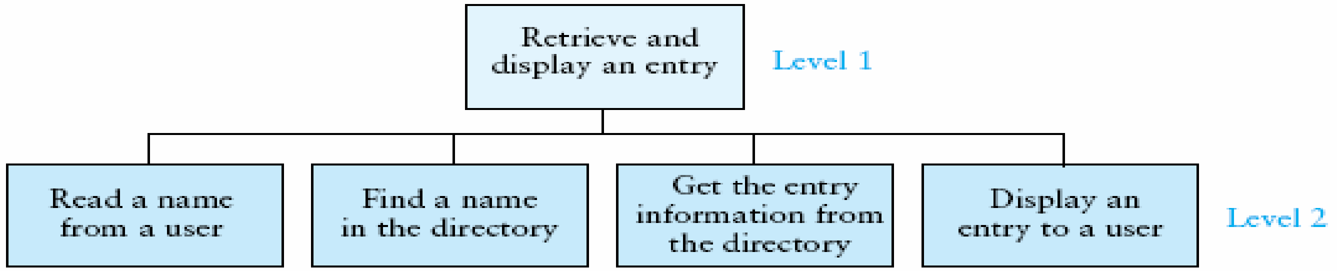
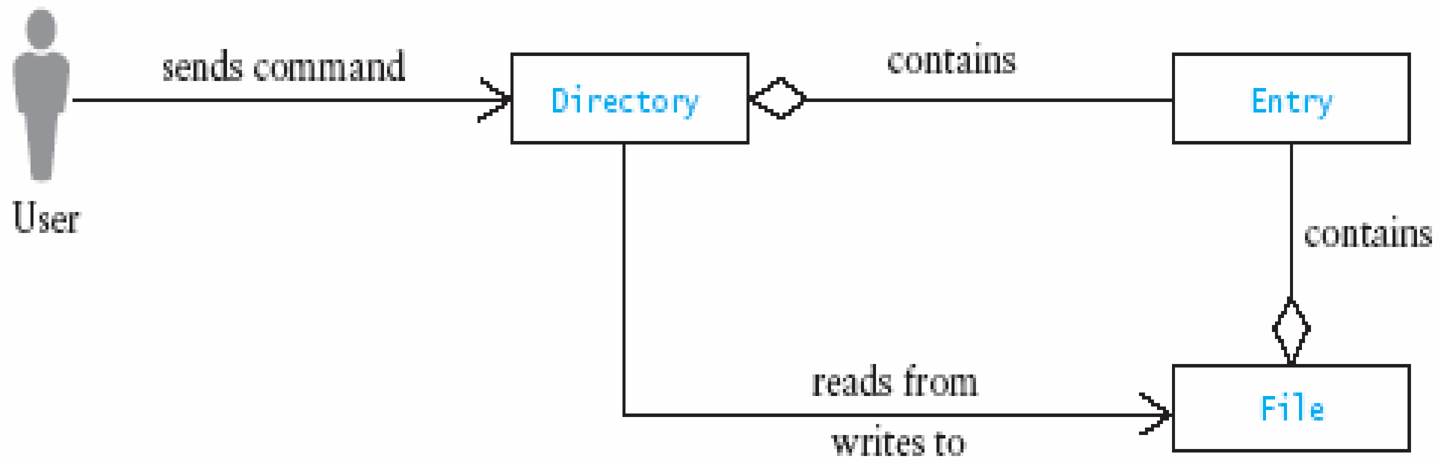


FIGURE 1.5
Refinement of "Retrieve and display an entry"


Retrieve and display an entry
Level 1
Read a name from a user
Find a name in the directory
Get the entry information from the directory
Display an entry to a user
Level 2

Example of Object-Oriented: Class Diagram

FIGURE 1.6
Initial Class Diagram for Phone Directory Program



Using Abstraction to Manage Complexity

- An ***abstraction*** is a model of a physical entity or activity
  - Models include relevant facts and details
  - Models exclude matters irrelevant to system/task
- Abstraction helps programmers:
  - Complex issues handled in manageable pieces
- **Procedural abstraction:** distinguishes ...
  - *What* to achieve (by a procedure) ...
  - From *how* to achieve it (implementation)
- **Data abstraction:** distinguishes ...
  - *Data objects* for a problem and their operations ...
  - From *their representation* in memory

Using Abstraction to Manage Complexity (2)

- If another class uses an object *only through its methods*, the other class will not be affected if the data representation changes
- **Information hiding:** Concealing the details of a class implementation from users of the class
  - Enforces the discipline of data abstraction

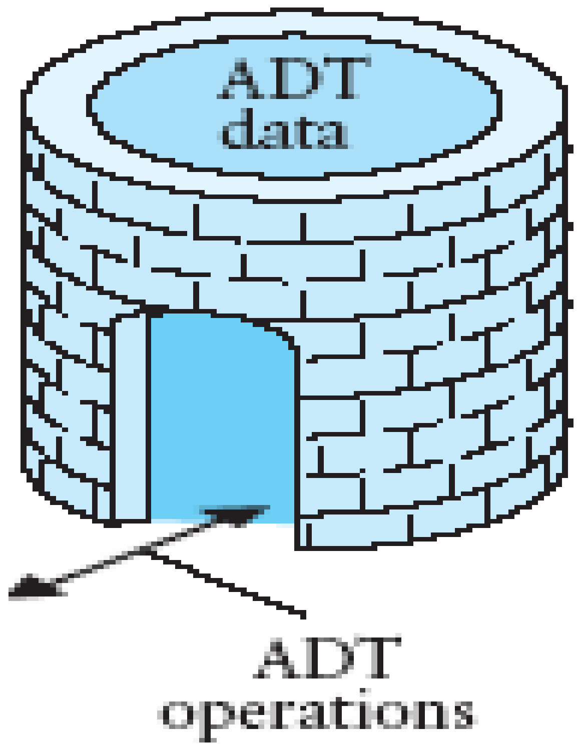
Abstract Data Types, Interfaces, and Pre- and Post-conditions

- A major goal of software engineering: *write reusable code*
- **Abstract data type** (ADT): data + methods
- A **Java interface** is a way to specify an ADT
  - Names, parameters, return types of methods
  - No indication of *how* achieved (procedural abstraction)
  - No representation (data abstraction)
- A class may ***implement*** an interface
  - Must provide bodies for all methods of the interface

Abstract Data Types, Interfaces, and Pre- and Postconditions (2)

FIGURE 1.7

Diagram of an ADT



Abstract Data Types, Interfaces, and Pre- and Postconditions (continued)

- You cannot *instantiate* (**new**) an interface
- But you *can:*
  - Declare a variable that has an interface type
  - Use it to reference an actual object, whose class implements the interface
- A Java interface is a *contract* between
  - The interface designer and ...
  - The coder of a class that implements the interface
- **Precondition:** any assumption/constraint on the method data before the method begins execution
- **Postcondition:** describes result of executing the method

Requirements Analysis: Use Cases, and Sequence Diagrams

- Analysis first step: study input and output requirements:
  - Make sure they are understood and make sense
- **Use case:**
  - User actions and system responses for a sub-problem
  - In the order that they are likely to occur
- **Sequence diagram:**
  - Shows objects involved across the horizontal axis
  - Shows time along the vertical axis
  - See page 26 for an example; shows:
    - User, PDApplication, PhoneDirectory, BufferedReader, PDUserInterface object + a number of method calls

Design of an Array-Based Phone Directory

- Case study shows:
  - Design
  - Implementation
  - Testing of a software-based phone directory

- In UML class diagrams:

  + sign next to a method/attribute means it is **`public`**

  - sign next to a method/attribute means it is **`private`**

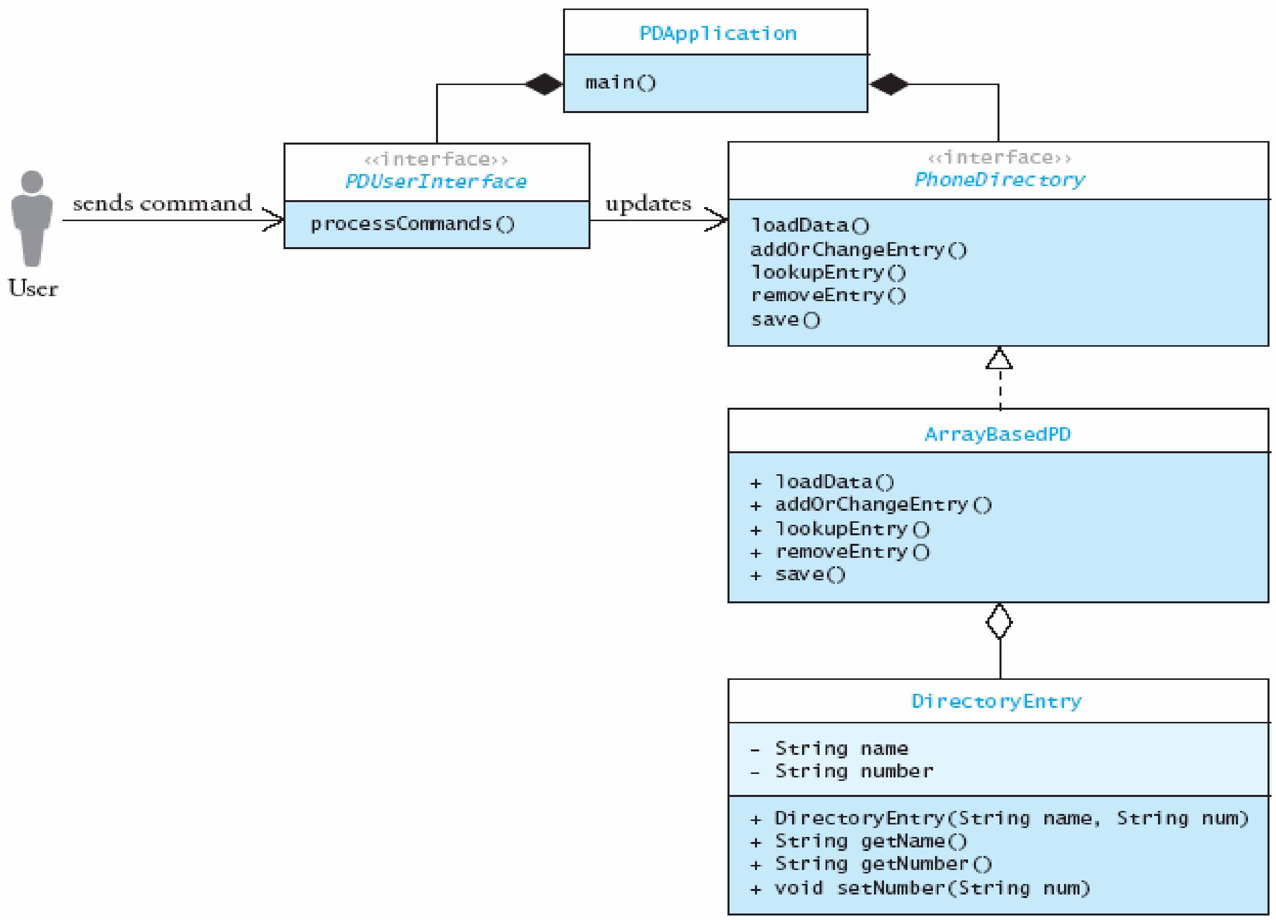
Design of Array-Based Phone Directory

Classes/interfaces to design include:

- ***`PDUserInterface`***: interface; later we consider:
    - Console (command line) UI class
    - Graphical (**`JOptionPane`**) UI class
- **`PDApplication`**: main / driving class
- ***`PhoneDirectory`***: interface
- **`ArrayBasedPD`**: class implementing **`PhoneDirectory`**
- **`DirectoryEntry`**: class, for one item in the directory

Design of Array-Based Phone Directory (2)

FIGURE 1.11
Phone Directory Application Class Diagram: Revision 2



Design of Array-Based Phone Directory (3)

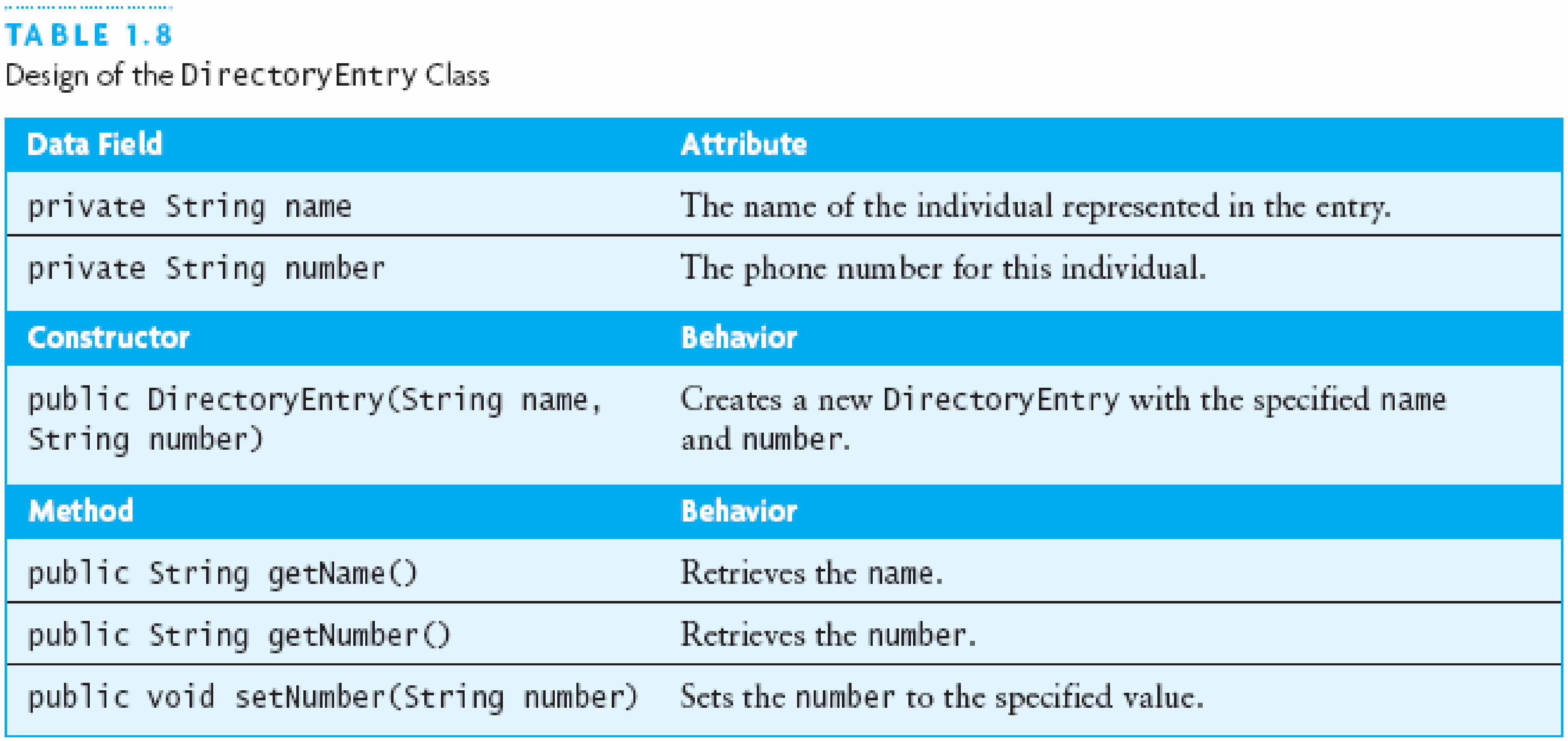


TABLE 1.8

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.

Design of **`DirectoryEntry`**

- Simple class, similar to **`Person`** in Java review:
  - Two private fields, for name and number
  - Two-argument constructor
  - Get methods for both fields
  - Set method for number (only)

Design of Array-Based Phone Directory (4)

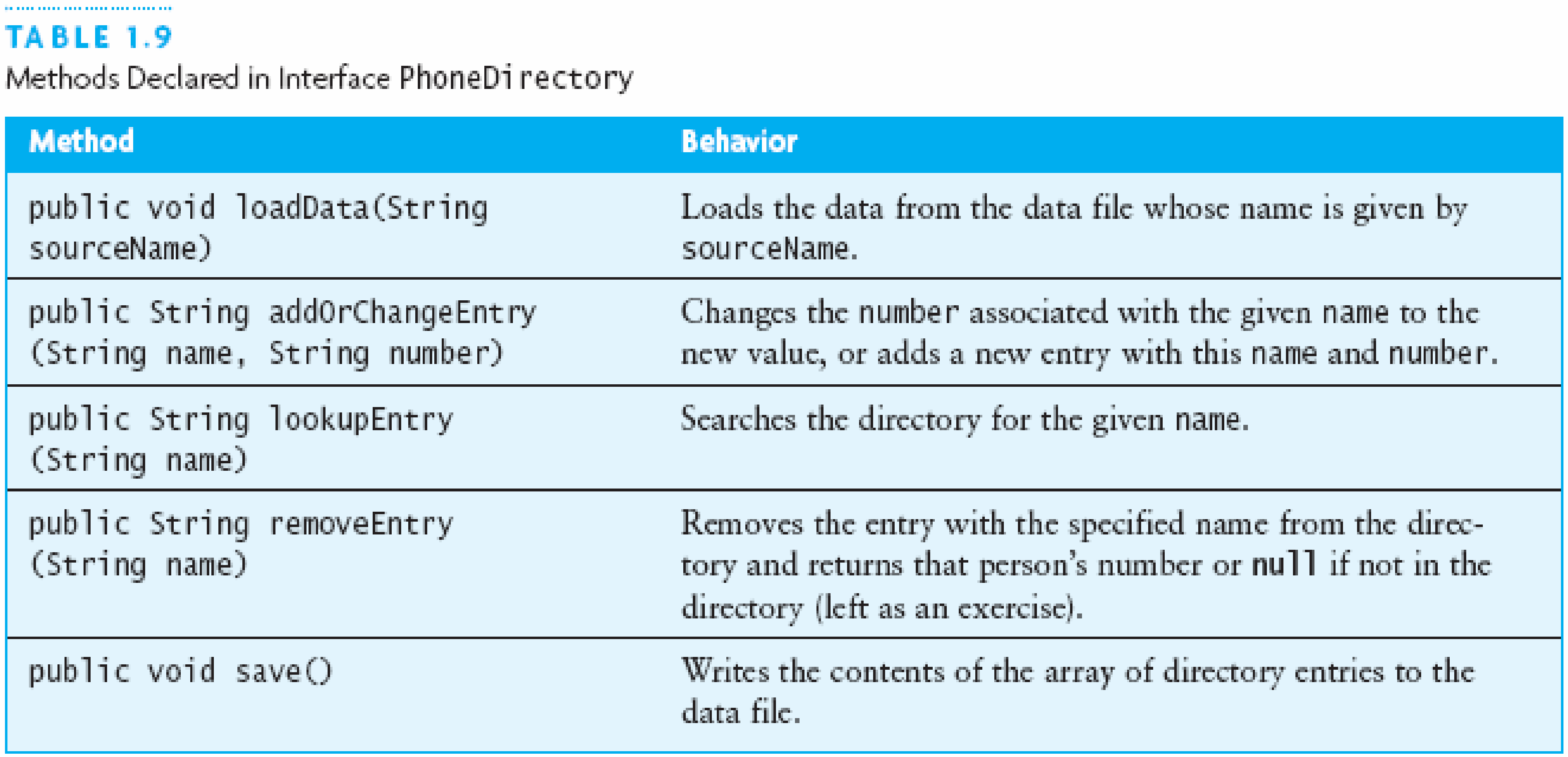


TABLE 1.9

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.

The **PhoneDirectory** Interface

```
/**
 * The interface for the telephone directory.
 * @author Koffman & Wolfgang
 */
public interface PhoneDirectory {
  ...
}
```

- Shows syntax of an **`interface`**
- Shows a javadoc comment and the **`@author`** tag

PhoneDirectory.loadData

```
/** 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);
```

- Shows syntax of method in an **`interface`** (note **`;`**)
- Shows a javadoc comment with the **`@param`** tag
- Since returns **`void`**, no **`@return`** tag

PhoneDirectory.lookupEntry

```
/** 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);
```

- Shows a javadoc comment with the **`@return`** tag
- I prefer a space before the (in a declaration (not a call)

PhoneDirectory.addOrChangeEntry

```
/** 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);
```

- Shows a javadoc comment with two **`@param`** tags

PhoneDirectory.removeEntry

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

PhoneDirectory.save

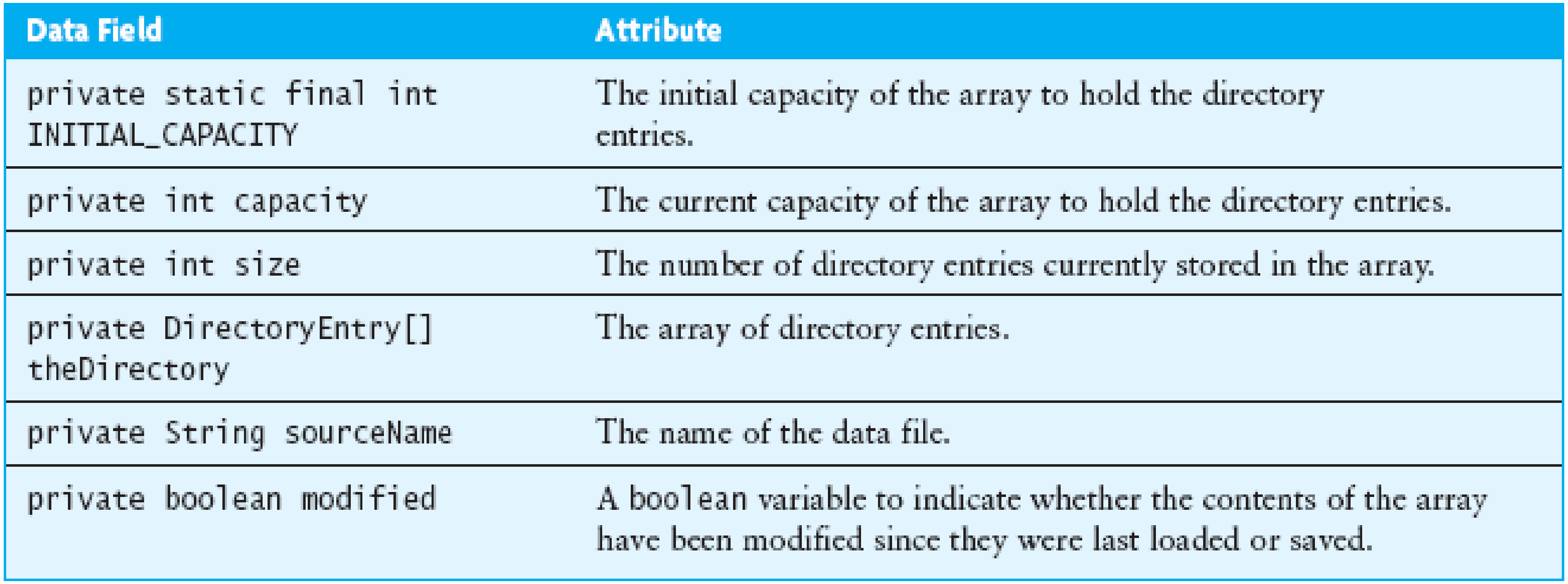
```
/** Method to save the directory.
 * pre:  The directory is loaded with data.
 * post: Contents of directory written back to
 *   the file in the form of name-number pairs
 *   on adjacent lines;
 *   modified is reset to false.
 */
void save ();
```

- Illustrates pre/post conditions

Design of Array-Based Phone Directory (5)

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.loadData`**

Input: a file name; Effect: read initial directory from the file

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

Design of **`ArrayBasedPD.addOrChangeEntry`**

Input: name and number; Effect: change number of existing entry, or make new entry if there was none

```
1.  Call method find to see if the name is in the directory
2.   if the name is in the directory
3.          change number with DirectoryEntry.setNumber
4.          Return the previous value of the number
     else
5.          Add a new entry using method add
6.          Return null
```

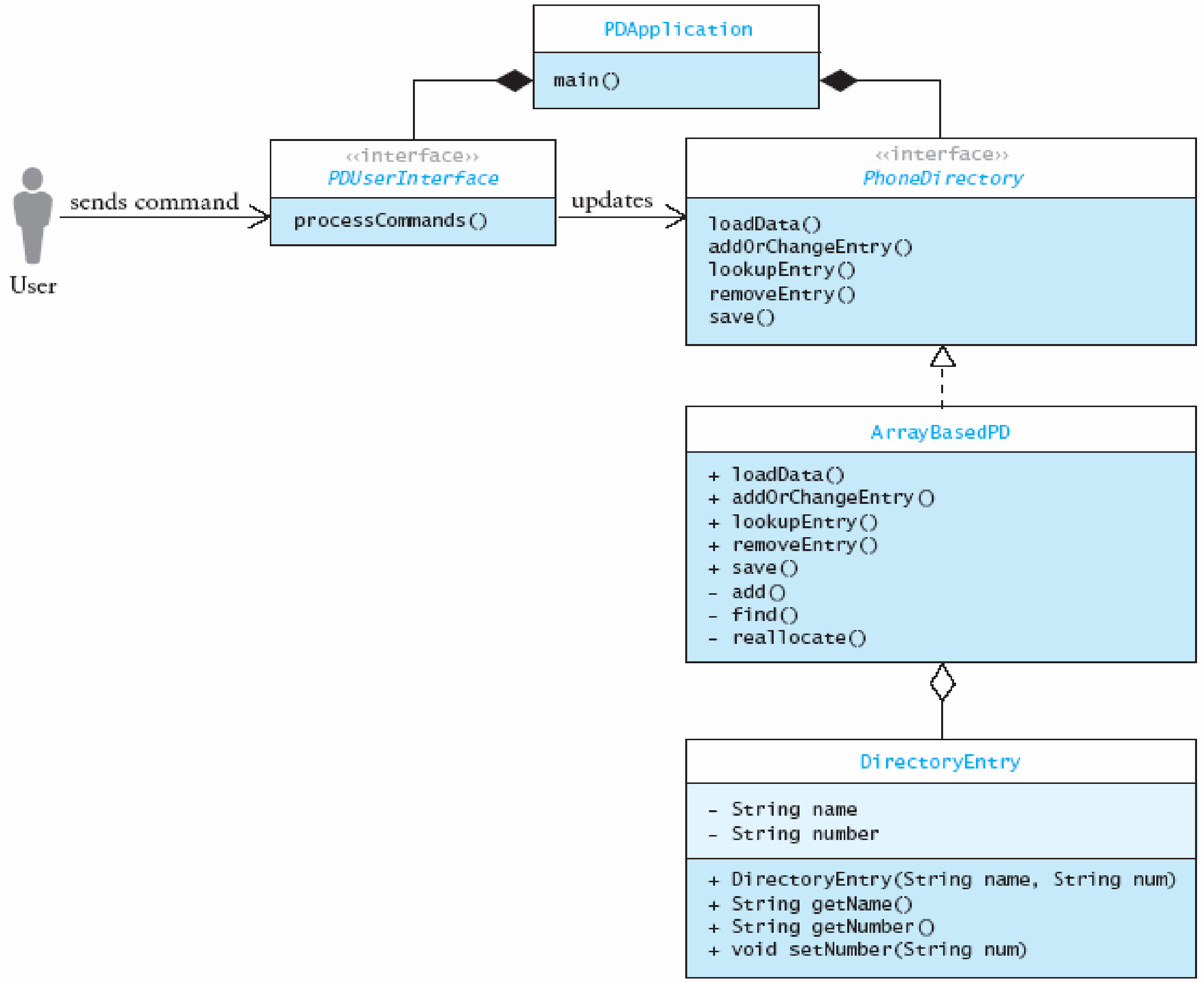
Design of Array-Based Phone Directory (6)

- Remaining method designs proceed along the same lines
- The class diagram changes, showing private fields and methods added ....

Design of Array-Based Phone Directory (7)

FIGURE 1.12

Phone Directory Application Class Diagram: Revision 3



Implementing and Testing the Array-Based Phone Directory: **`ArrayBasedPD.java`**

```
import java.io.*;
/** This is an implementation of the
 *  PhoneDirectory interface that uses an
 *  array to store the data.
 *  @author Koffman & Wolfgang
 */
public class ArrayBasedPD
    implements PhoneDirectory {
  ...
}   // note: import, javadoc, implements
```

`ArrayBasedPD` Data Fields (1)

```
// Data Fields       (with javadoc comments)
/** 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;
```

ArrayBasedPD Data Fields (2)

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

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

/** Boolean flag indicates if the directory was
    modified since it was loaded or saved. */
private boolean modified = false;
```

ArrayBasedPD.loadData

```
public void loadData (String sourceName) {
  // Remember the source name.
  this.sourceName = sourceName;
  try {  ...
  } catch (FileNotFoundException ex) {
    // Do nothing – no data to load.
    return;
  } catch (IOException ex) {
    System.err.println(“Directory load failed.");
    ex.printStackTrace();
    System.exit(1);
  }
}
```

`ArrayBasedPD.loadData` (2): Inside **`try`**

```
BufferedReader in = new BufferedReader(
  new FileReader(sourceName));
while (true) {
  String name, number;
  // read name and number from succeeding lines
  if ((name   = in.readLine()) == null) break;
  if ((number = in.readLine()) == null) break;
  // insert entry (if got both name and number)
  add(name, number);
}
in.close();   // should always close input
```

- Slightly different loop approach from the text
- Same assign-in-**`if`**-condition “hack”

`ArrayBasedPD.loadData` (3): alternate

```
boolean more = true;
while (more) {
  more = false;
  String name = in.readLine();
  if (name != null) {
    String number = in.readLine();
    if (number != null) {
      add(name, number);
      more = true;
    }
  }
}
```

- Nested **`if`** statements not as pleasant (what if 7 inputs?)
- Control variables tend to be harder to understand/get right

ArrayBasedPD.addOrChangeEntry

```
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;
}
```

ArrayBasedPD.save

```
public void save() {
  if (!modified) return; // save not needed
  try {
    // Create PrintWriter for the file.
    PrintWriter out = new PrintWriter(
        new FileWriter(sourceName));
    ...
} catch (Exception ex) {
    System.err.println(“Directory save failed");
    ex.printStackTrace();
    System.exit(1);
  }
}
```

ArrayBasedPD.save (2)

```
// 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;
```

Implementing and Testing the Array-Based Phone Directory

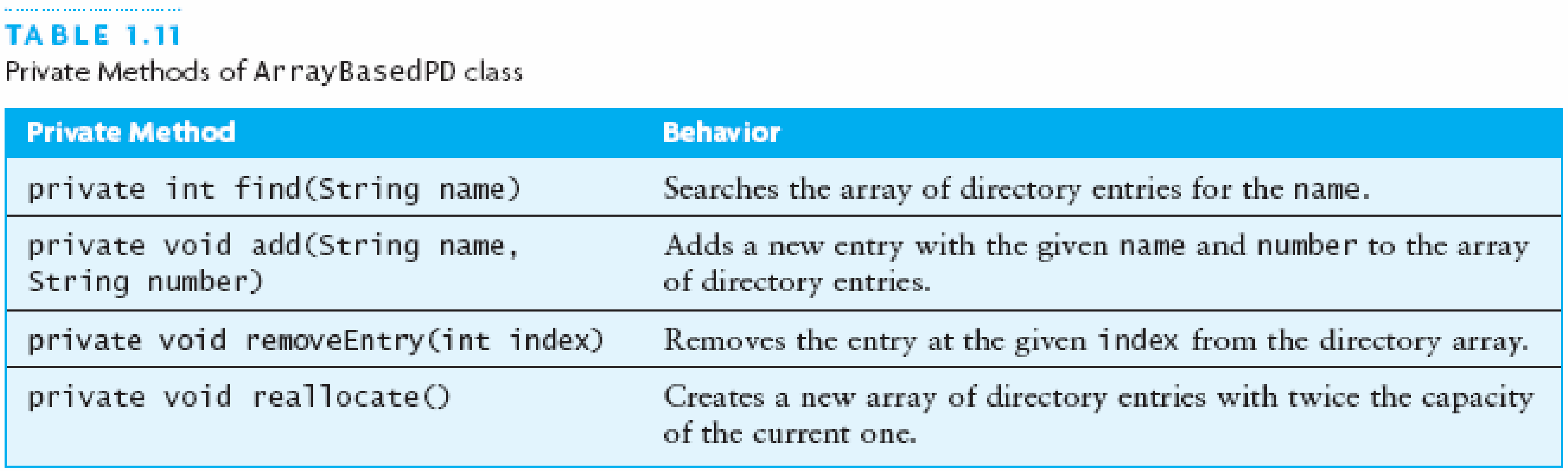


TABLE 1.11
Private Methods of ArrayBasedPD class

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.

ArrayBasedPD.find

```
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.
}
```

ArrayBasedPD.add

```
private void add (String name, String number) {
  if (size >= capacity) {
    reallocate();
  }
  theDirectory[size++] =
    new DirectoryEntry(name, number);
}
```

- Differs from text in use of `++`
- Note that `size` means number of names stored,
- while `capacity` means the number the array can hold

ArrayBasedPD.realloc

```
private void reallocate () {
  capacity *= 2;
  DirectoryEntry[] newDirectory =
    new DirectoryEntry[capacity];
  System.arraycopy(theDirectory, 0,
                   newDirectory, 0,
                   theDirectory.length);
  theDirectory = newDirectory;
}
```

Arguments to **`arraycopy`** are:

- fromDir, fromIndex
- toDir, toIndex
- number of elements to copy

Testing `ArrayBasedPD`

- Empty data file
- Data file with only one name-number pair
- Data file with odd number of lines
- Data file with more pairs than initial array size
- Retrieve names *not* in directory as well as ones that are
- After a change, verify the new information
- Check that after changes, the changes, plus all new information, are in the newly written file

- Note: This code does not check for empty strings!

Implementing **`PDUserInterface`**

- Text offers two *classes* that implement the UI *interface:*
  - **`PDGUI`**: Uses **`JOptionPane`** for graphical UI
  - **`PDConsoleUI`**: Uses console stream I/O (**`System.in`** and **`System.out`**)
- Text gives good recipes here that you can use as models
- We will not cover them in detail here