Object-Oriented and Classical Software Engineering

FROM MODULES TO OBJECTS

Overview

- What is a module?
- Cohesion
- Coupling
- Data encapsulation
- Abstract data types
- Information hiding
- Objects
- Inheritance, polymorphism, and dynamic binding
- The object-oriented paradigm

7.1 What Is a Module?

- A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
  - "Lexically contiguous"
    - Adjoining in the code
  - "Boundary elements"
    - `{ ... }`
    - **`begin ... end`**
  - "Aggregate identifier"
    - A name for the entire module

Design of Computer

- A highly incompetent computer architect decides to build an ALU, shifter, and 16 registers with AND, OR, and NOT gates, rather than NAND or NOR gates

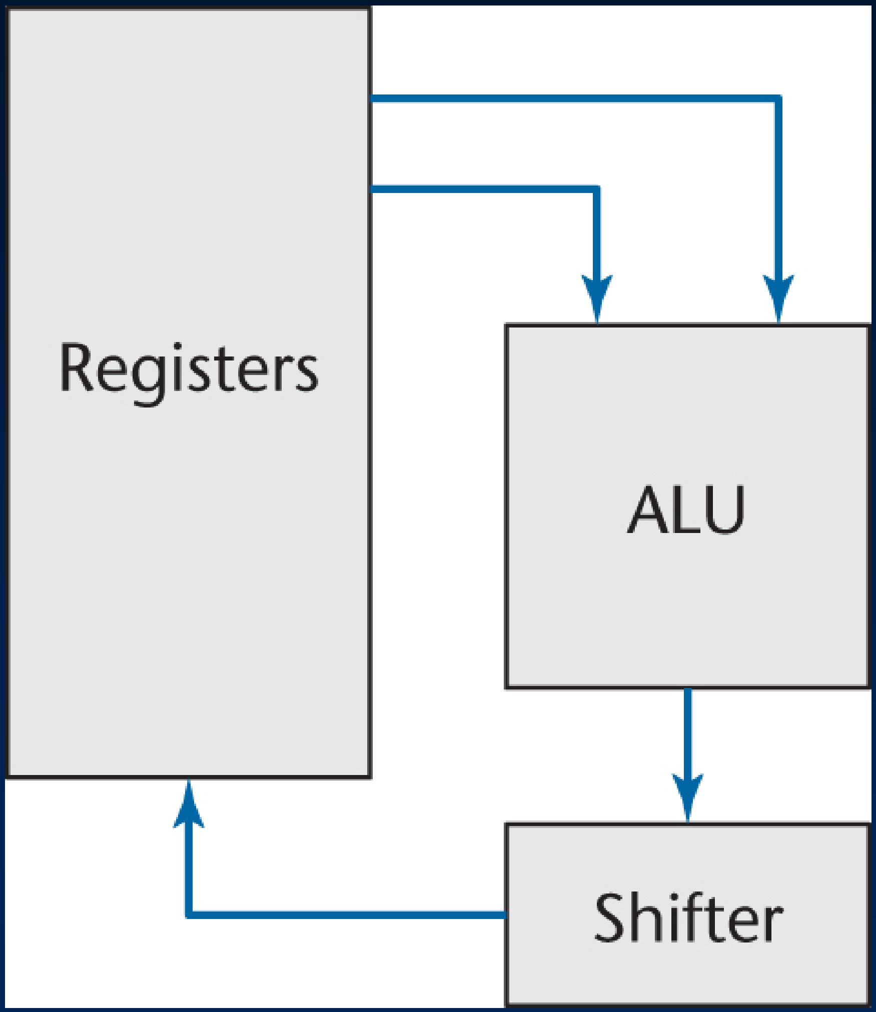

Figure 7.1

Design of Computer (contd)

- The architect designs three silicon chips

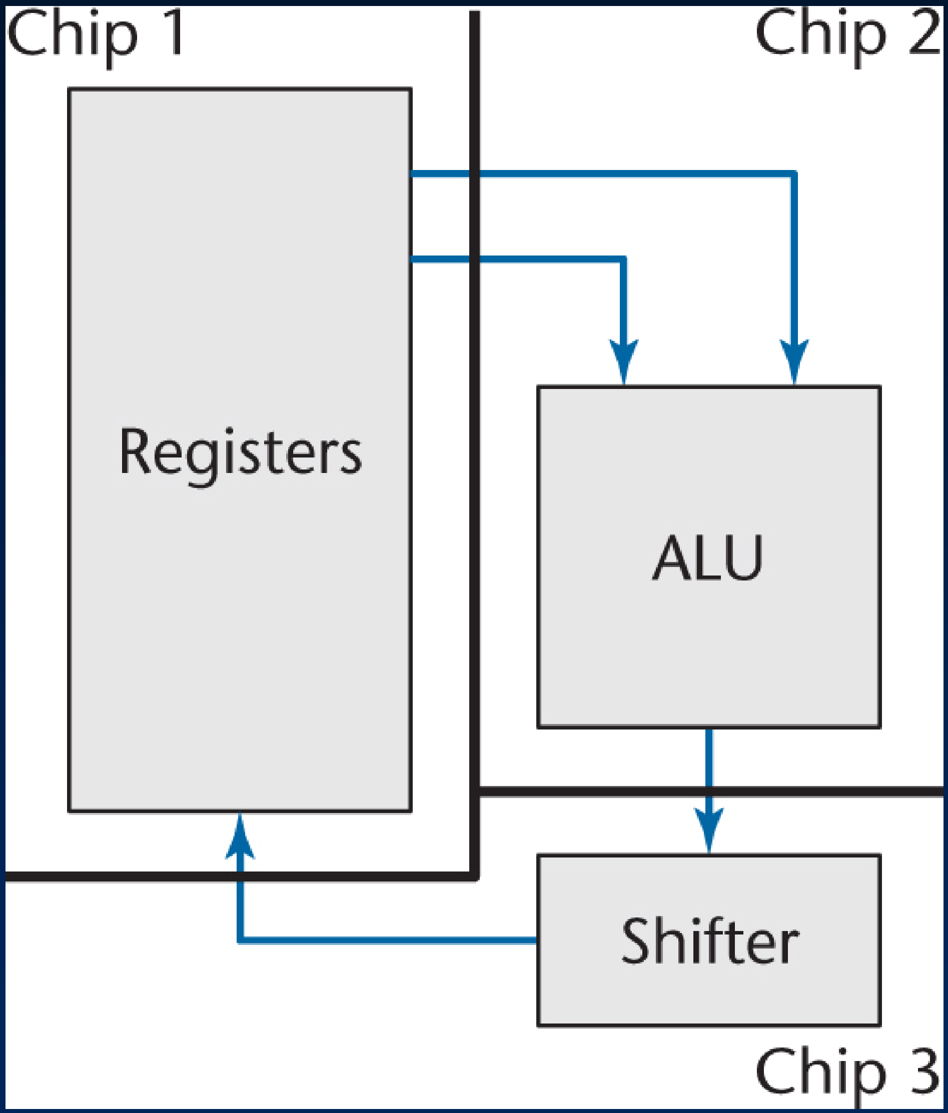

Figure 7.2

Design of Computer (contd)

- Redesign with one gate type per chip
- Resulting "masterpiece"

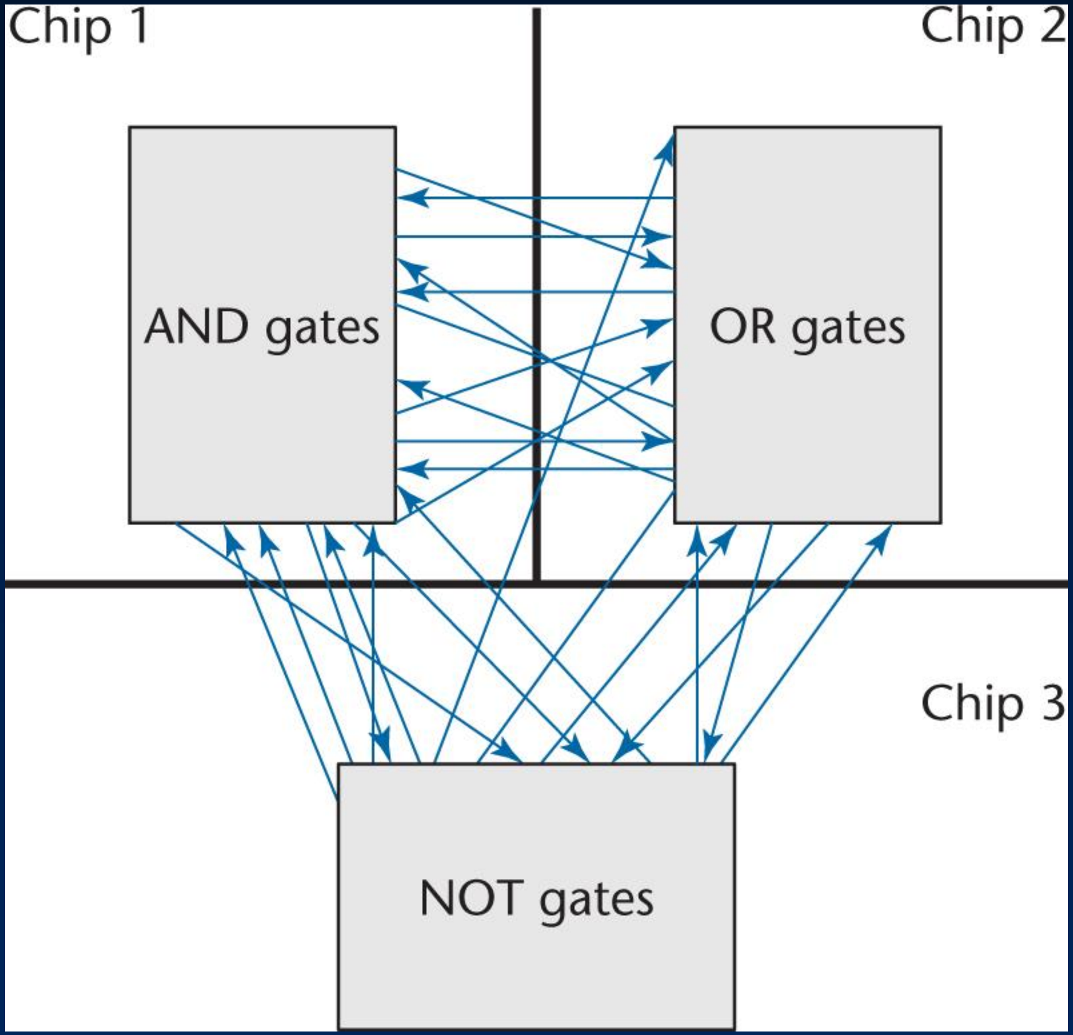

Figure 7.3

Computer Design (contd)

- The two designs are functionally equivalent
  - The second design is
    - » Hard to understand
    - » Hard to locate faults
    - » Difficult to extend or enhance
    - » Cannot be reused in another product

- Modules must be like the first design
  - Maximal relationships within modules, and
  - Minimal relationships between modules

Composite/Structured Design

- A method for breaking up a product into modules to achieve
  - Maximal interaction within a module, and
  - Minimal interaction between modules
- Module cohesion
  - Degree of interaction within a module
- Module coupling
  - Degree of interaction between modules

Function, Logic, and Context of a Module

- In C/SD, the name of a module is its function

- Example:
  - A module computes the square root of double precision integers using Newton's algorithm. The module is named `compute_square_root`

- The underscores denote that the classical paradigm is used here

7.2 Cohesion

- The degree of interaction within a module
- Seven categories or levels of cohesion (non-linear scale)

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Figure 7.4

7.2.1 Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions

- Example:
  - `print_next_line, reverse_string_of_characters_comprising_second_parameter, add_7_to_fifth_parameter, convert_fourth_parameter_to_floating_point`

- Such modules arise from rules like
  - “Every module will consist of between 35 and 50 statements”

Why Is Coincidental Cohesion So Bad?

- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
  - Break the module into separate modules, each performing one task

7.2.2 Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module

Logical Cohesion (contd)

- Example 1:

```
function_code = 7;
new_operation (functin_code, dummy_1, dummy_2, dummy_3);
// dummy_1, dummy_2, and dummy_3 are dummy variables,
// not used if function code is equal to 7
```

- Example 2:
  - An object performing all input and output
- Example 3:
  - One version of OS/VS2 contained a module with logical cohesion performing 13 different actions. The interface contains 21 pieces of data

Why Is Logical Cohesion So Bad?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

Why Is Logical Cohesion So Bad? (contd)

- A new tape unit is installed
  - What is the effect on the laser printer?

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

Figure 7.5

7.2.3 Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time

- Example:
  - `open_old_master_file, new_master_file, transaction_file, and print_file; initialize_sales_district_table, read_first_transaction_record, read_first_old_master_record (a.k.a. perform_initialization)`

Why Is Temporal Cohesion So Bad?

- The actions of this module are weakly related to one another, but strongly related to actions in other modules
  - Consider `sales_district_table`
- Not reusable

7.2.4 Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product

- Example:
  - `read_part_number_and_update_repair_record_on_master_file`

Why Is Procedural Cohesion So Bad?

- The actions are still weakly connected, so the module is not reusable

7.2.5 Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data

- Example 1:

```
update_record_in_database_and_write_it_to_audit_trail
```

- Example 2:

```
calculate_new_coordinates_and_send_them_to_terminal
```

Why Is Communicational Cohesion So Bad?

- Still lack of reusability

7.2.6 Functional Cohesion

- A module with functional cohesion performs exactly one action

7.2.6 Functional Cohesion

- Example 1:
  - `get_temperature_of_furnace`
- Example 2:
  - `compute_orbital_of_electron`
- Example 3:
  - `write_to_diskette`
- Example 4:
  - `calculate_sales_commission`

Why Is Functional Cohesion So Good?

- More reusable
- Corrective maintenance is easier
  - Fault isolation
  - Fewer regression faults
- Easier to extend a product

7.2.7 Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

Why Is Informational Cohesion So Good?

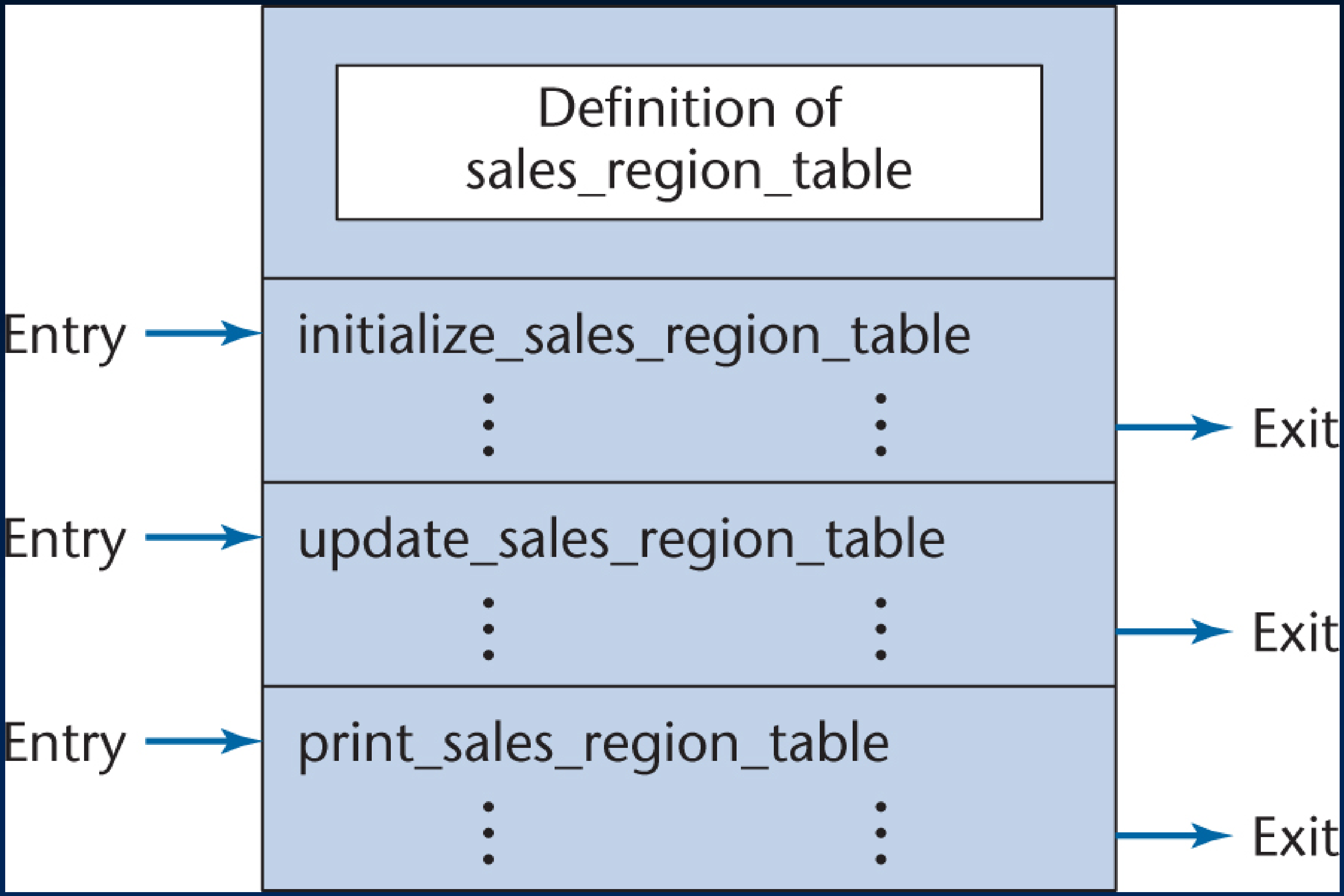

Figure 7.6

- Essentially, this is an abstract data type (see later)

7.2.8 Cohesion Example

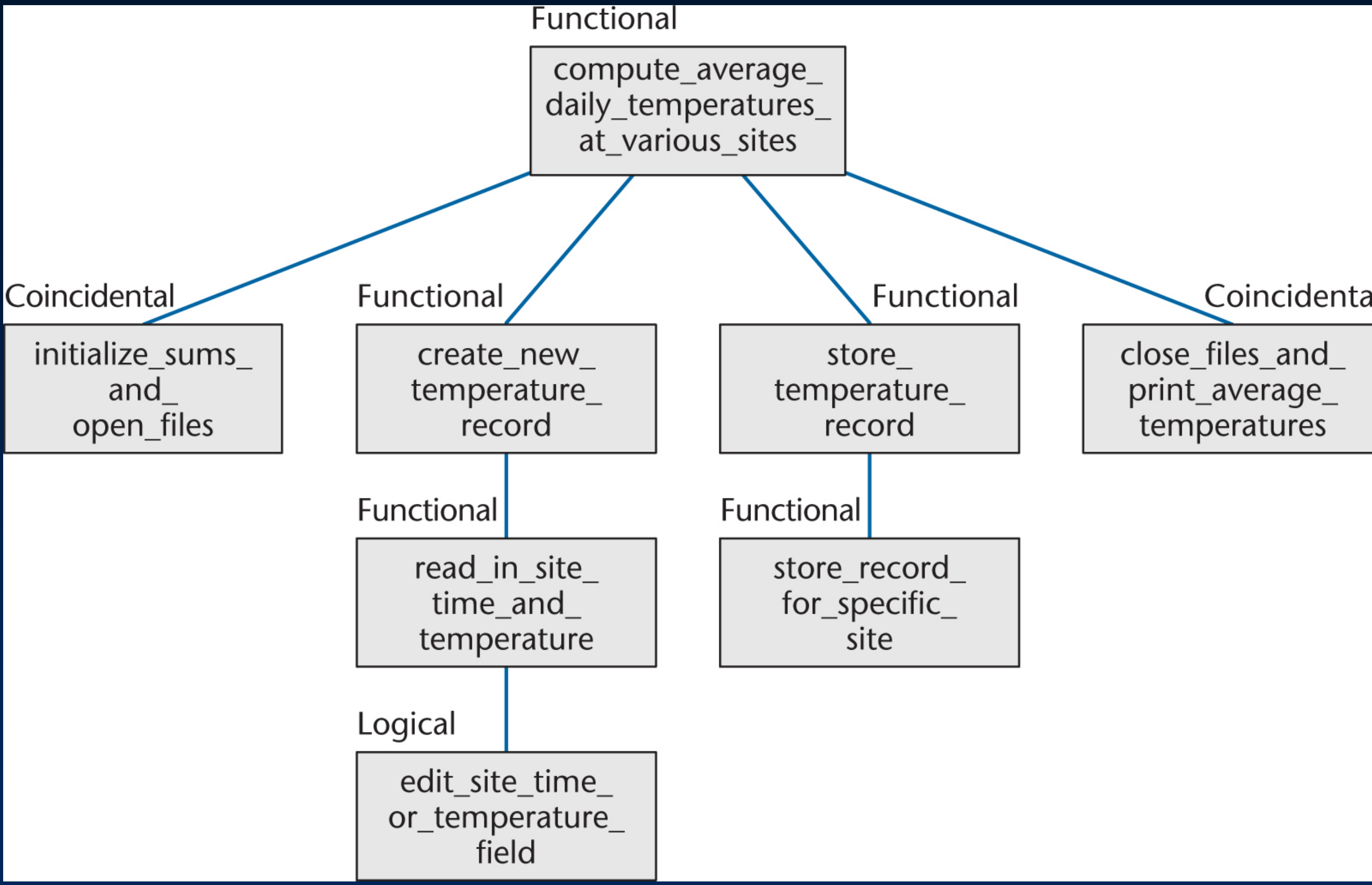

Figure 7.7

7.3 Coupling

- The degree of interaction between two modules
  - Five categories or levels of coupling (non-linear scale)

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Figure 7.8

7.3.1 Content Coupling

- Two modules are content coupled if one directly references contents of the other

- Example 1:
  - Module `p` modifies a statement of module `q`

- Example 2:
  - Module `p` refers to local data of module `q` in terms of some numerical displacement within `q`

- Example 3:
  - Module `p` branches into a local label of module `q`

Why Is Content Coupling So Bad?

- Almost any change to module `q`, even recompiling `q` with a new compiler or assembler, requires a change to module `p`

7.3.2 Common Coupling

- Two modules are common coupled if they have write access to global data

Figure 7.9

- Example 1
  – Modules `cca` and `ccb` can access *and change* the value of `global_variable`

7.3.2 Common Coupling (contd)

- Example 2:
  - Modules `cca` and `ccb` both have access to the same database, and can both read *and write* the same record

- Example 3:
  - FORTRAN **`common`**
  - COBOL **`common`** (nonstandard)
  - COBOL-80 **`global`**

Why Is Common Coupling So Bad?

- It contradicts the spirit of structured programming
  - The resulting code is virtually unreadable

```
while (global_variable == 0)
{
  if (argument_xyz > 25)
     module_3 ( );
  else
     module_4 ( );
}
```

Figure 7.10

  - What causes this loop to terminate?

Why Is Common Coupling So Bad? (contd)

- Modules can have side-effects
  - This affects their readability
  - Example: `edit_this_transaction (record_7)`
  - The entire module must be read to find out what it does

- A change during maintenance to the declaration of a global variable in one module necessitates corresponding changes in other modules

- Common-coupled modules are difficult to reuse

Why Is Common Coupling So Bad? (contd)

- Common coupling between a module `p` and the rest of the product can change without changing `p` in any way
  - *Clandestine common coupling*
  - Example: The Linux kernel
- A module is exposed to more data than necessary
  - This can lead to computer crime

7.3.3 Control Coupling

- Two modules are control coupled if one passes an element of control to the other

- Example 1:
  - An operation code is passed to a module with logical cohesion

- Example 2:
  - A control switch passed as an argument

Control Coupling (contd)

- Module `p` calls module `q`
- Message:
  - `I have failed` — data
- Message:
  - `I have failed, so write error message ABC123` — control

Why Is Control Coupling So Bad?

- The modules are not independent
  - Module `q` (the called module) must know the internal structure and logic of module `p`
  - This affects reusability
- Associated with modules of logical cohesion

7.3.4 Stamp Coupling

- Some languages allow only simple variables as parameters
  - `part_number`
  - `satellite_altitude`
  - `degree_of_multiprogramming`
- Many languages also support the passing of data structures
  - `part_record`
  - `satellite_coordinates`
  - `segment_table`

Stamp Coupling (contd)

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Why Is Stamp Coupling So Bad?

- It is not clear, without reading the entire module, which fields of a record are accessed or changed
  - Example

    ```
    calculate_withholding (employee_record)
    ```

- Difficult to understand
- Unlikely to be reusable
- More data than necessary is passed
  - Uncontrolled data access can lead to computer crime

Why Is Stamp Coupling So Bad? (contd)

- However, there is nothing wrong with passing a data structure as a parameter, provided that *all* the components of the data structure are accessed and/or changed

- Examples:

```
invert_matrix (original_matrix, inverted_matrix);
print_inventory_record (warehouse_record);
```

7.3.5 Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items (simple parameters, or data structures all of whose elements are used by called module)

- Examples:
  - `display_time_of_arrival (flight_number);`
  - `compute_product (first_number, second_number);`
  - `get_job_with_highest_priority (job_queue);`

Why Is Data Coupling So Good?

- The difficulties of content, common, control, and stamp coupling are not present

- Maintenance is easier

7.3.6. Coupling Example

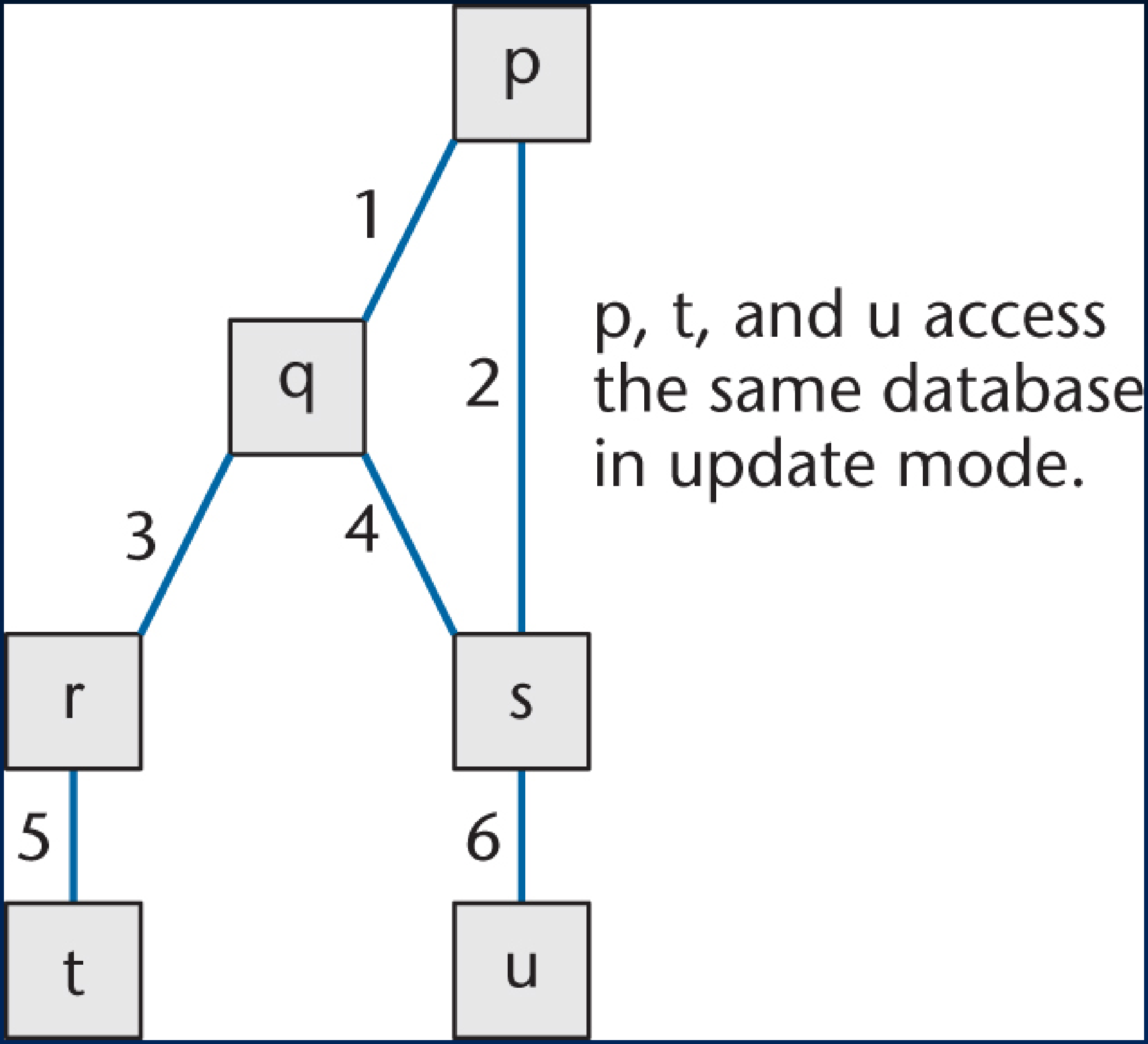

Figure 7.11

Coupling Example (contd)

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

Figure 7.12

- Interface description

Coupling Example (contd)

	q	r	s	t	u
p	Data	—	Data or stamp	Common	Common
q		Control	Data or stamp	—	—
r			—	Data	—
s				—	Data
t					Common

Figure 7.13

- Coupling between all pairs of modules

7.3.7 The Importance of Coupling

- As a result of tight coupling
  - A change to module `p` can require a corresponding change to module `q`
  - If the corresponding change is not made, this leads to faults
- Good design has high cohesion and low coupling
  - What else characterizes good design? (see over)

Key Definitions

Abstract data type: a data type together with the operations performed on instantiations of that data type (Section 7.5)

Abstraction: a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details (Section 7.4.1)

Class: an abstract data type that supports inheritance (Section 7.7)

Cohesion: the degree of interaction within a module (Section 7.1)

Coupling: the degree of interaction between two modules (Section 7.1)

Data encapsulation: a data structure together with the operations performed on that data structure (Section 7.4)

Encapsulation: the gathering together into one unit of all aspects of the real-world entity modeled by that unit (Section 7.4.1)

Information hiding: structuring the design so that the resulting implementation details are hidden from other modules (Section 7.6)

Object: an instantiation of a class (Section 7.7)

Figure 7.14

7.4 Data Encapsulation

- Example
  - Design an operating system for a large mainframe computer. Batch jobs submitted to the computer will be classified as high priority, medium priority, or low priority. There must be three queues for incoming batch jobs, one for each job type. When a job is submitted by a user, the job is added to the appropriate queue, and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it

- Design 1 (Next slide)
  - Low cohesion — operations on job queues are spread all over the product

Data Encapsulation — Design 1

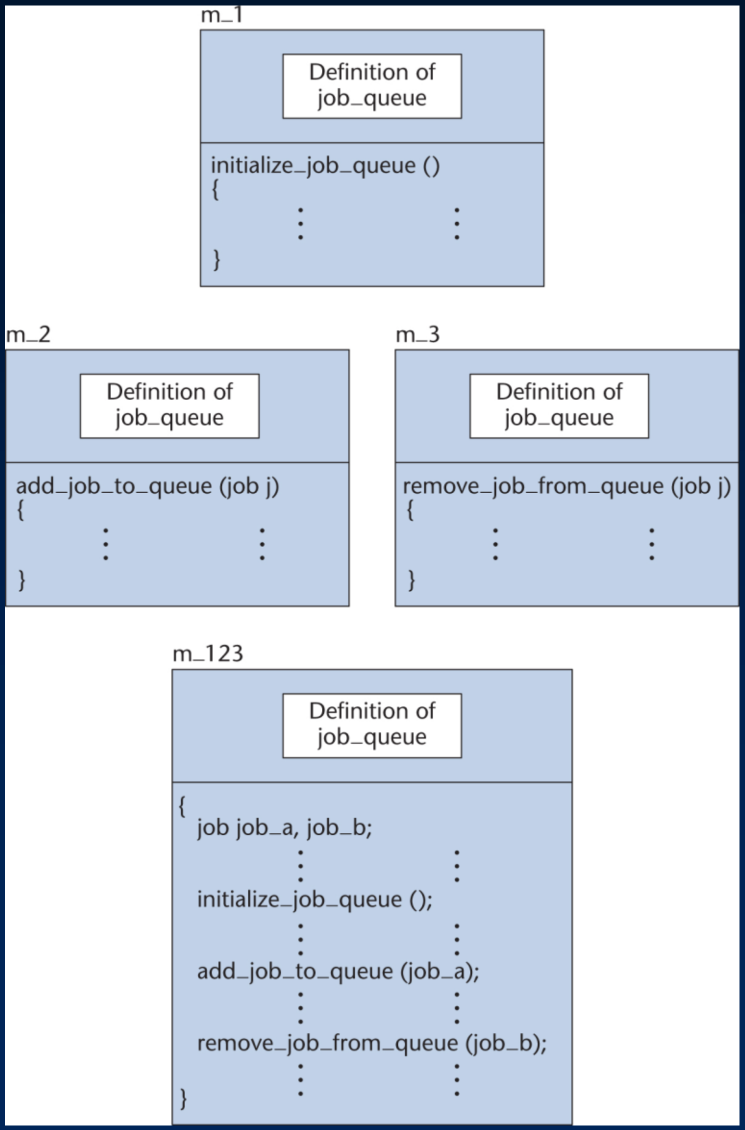

Figure 7.15

Data Encapsulation — Design 2

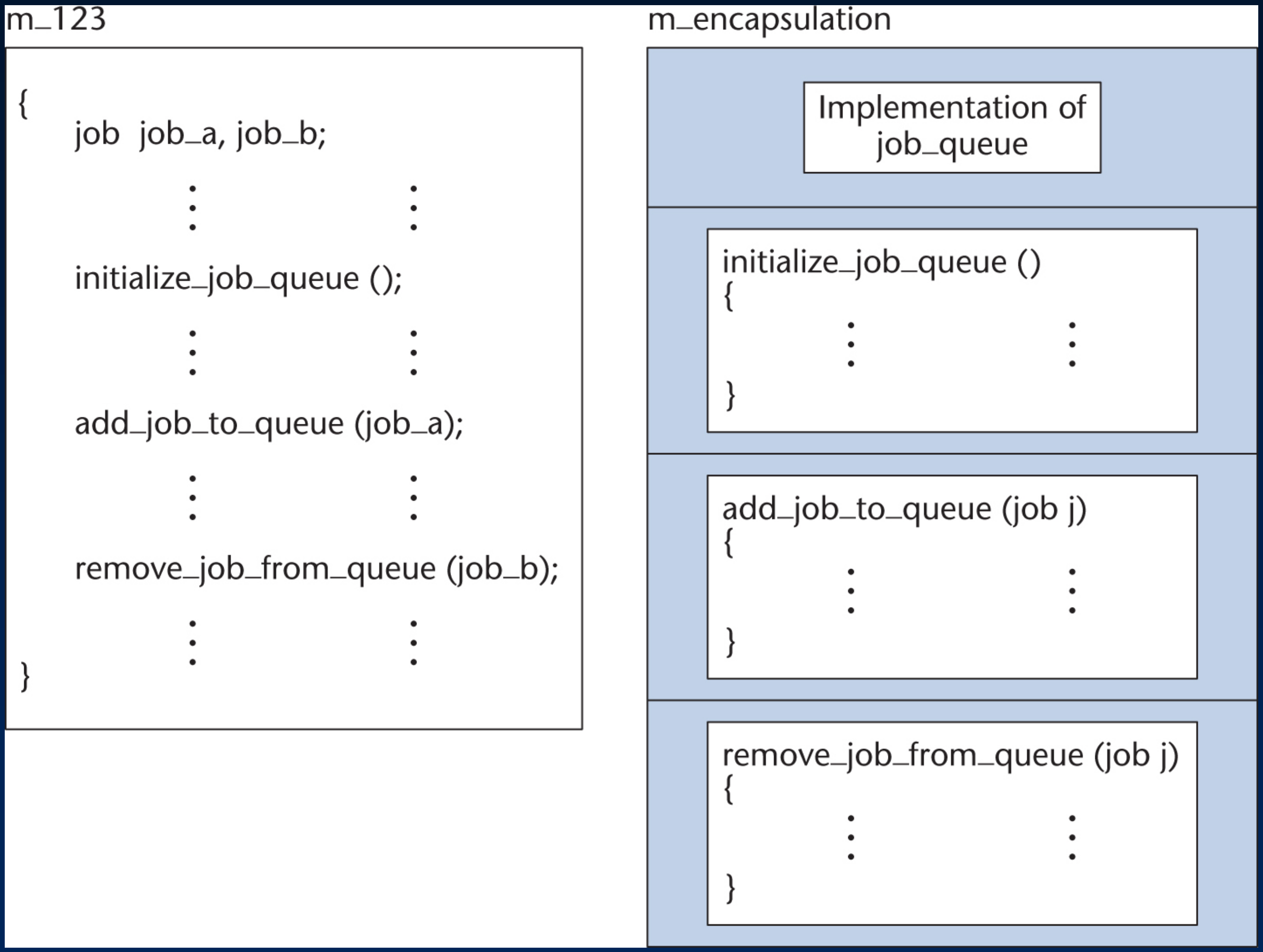

Figure 7.16

Data Encapsulation (contd)

- `m_encapsulation` has informational cohesion

- `m_encapsulation` is an implementation of data encapsulation
  - A data structure (`job_queue`) together with operations performed on that data structure

- Advantages
  - Development
  - Maintenance

Data Encapsulation and Development

- Data encapsulation is an example of *abstraction*
- Job queue example:
  - Data structure
    - » `job_queue`
  - Three new functions
    - » `initialize_job_queue`
    - » `add_job_to_queue`
    - » `delete_job_from_queue`

7.4.1 Data Encapsulation and Development

- Abstraction
  - Conceptualize problem at a higher level
    - Job queues and operations on job queues
  - Not a lower level
    - Records or arrays

Stepwise Refinement

1. Design the product in terms of higher level concepts
   - It is irrelevant how job queues are implemented

2. Then design the lower level components
   - Totally ignore what use will be made of them

Stepwise Refinement (contd)

- In the 1st step, assume the existence of the lower level
  - Our concern is the behavior of the data structure
    - `job_queue`
- In the 2nd step, ignore the existence of the higher level
  - Our concern is the implementation of that behavior
- In a larger product, there will be many levels of abstraction

7.4.2 Data Encapsulation and Maintenance

- Identify the aspects of the product that are likely to change

- Design the product so as to minimize the effects of change
  - Data structures are unlikely to change
  - Implementation details may change

- Data encapsulation provides a way to cope with change

Implementation of `JobQueueClass`

C++

```
//
// Warning:
// This code has been implemented in such a way as to be accessible to readers
// who are not C++ experts, as opposed to using good C++ style. Also, vital
// features such as checks for overflow and underflow have been omitted for simplicity.
// See Just in Case You Wanted to Know Box 7.3 for details.
//
class JobQueueClass
{
  // attributes
  public:
    int queueLength;      // length of job queue
    int queue[25];        // queue can contain up to 25 jobs

  // methods
  public:
    void initializeJobQueue ( )
    /*
     * an empty job queue has length 0
     */
    {
      queueLength = 0;
    }


    void addJobToQueue (int jobNumber)
    /*
     * add the job to the end of the job queue
     */
    {
      queue[queueLength] = jobNumber;
      queueLength = queueLength + 1;
    }


    int removeJobFromQueue ( )
    /*
     * set jobNumber equal to the number of the job stored at the head of the queue,
     * remove the job at the head of the job queue, move up the remaining jobs,
     * and return jobNumber
     */
    {
      int jobNumber = queue[0];
      queueLength = queueLength – 1;
      for (int k = 0; k < queueLength; k++)
        queue[k] = queue[k + 1];
      return jobNumber;
    }
}// class JobQueueClass
```

Figure 7.17

Java

```
//
// Warning:
// This code has been implemented in such a way as to be accessible to readers
// who are not Java experts, as opposed to using good Java style.
// Also, vital features such as checks for overflow and underflow
// have been omitted for simplicity.
// See Just in Case You Wanted to Know Box 7.3 for details.
//
class JobQueueClass
{
  // attributes
  public int    queueLength;               // length of job queue
  public int    queue[ ] = new int[25];    // queue can contain up to 25 jobs

  // methods
  public void initializeJobQueue ( )
  /*
   * an empty job queue has length 0
   */
  {
    queueLength = 0;
  }


  public void addJobToQueue (int jobNumber)
  /*
   * add the job to the end of the job queue
   */
  {
    queue[queueLength] = jobNumber;
    queueLength = queueLength + 1;
  }


  public int removeJobFromQueue ( )
  /*
   * set jobNumber equal to the number of the job stored at the head of the queue,
   * remove the job at the head of the job queue, move up the remaining jobs,
   * and return jobNumber
   */
  {
    int jobNumber = queue[0];
    queueLength = queueLength – 1;
    for (int k = 0; k < queueLength; k++)
      queue[k] = queue[k + 1];
    return jobNumber;
  }
}// class JobQueueClass
```

Figure 7.18

Implementation of `queueHandler`

C++

```
class SchedulerClass
{
  . . .
  public:
    void queueHandler ( )
    {
      int              jobA, jobB;
      JobQueueClass    jobQueueJ;

          // various statements
      jobQueueJ.initializeJobQueue ( );
          // more statements
      jobQueueJ.addJobToQueue (jobA);
          // still more statements
      jobB = jobQueueJ.removeJobFromQueue ( );
          // further statements
    }// queueHandler
  . . .
}// class SchedulerClass
```

Figure 7.19

Java

```
class SchedulerClass
{
  . . .
  public void queueHandler ( )
  {
    int              jobA, jobB;
    JobQueueClass    jobQueueJ; = new JobQueueClass ( );

        // various statements
    jobQueueJ.initializeJobQueue ( );
        // more statements
    jobQueueJ.addJobToQueue (jobA);
        // still more statements
    jobB = jobQueueJ.removeJobFromQueue ( );
        // further statements
  }// queueHandler
  . . .
}// class SchedulerClass
```

Figure 7.20

Data Encapsulation and Maintenance (contd)

- What happens if the queue is now implemented as a two-way linked list of `JobRecordClass`?
  - A module that uses `JobRecordClass` need not be changed at all, merely recompiled

C++

```
class JobRecordClass
{
  public:
        int                  jobNo;        // number of the job (integer)
        JobRecordClass       *inFront;     // pointer to the job record in front
        JobRecordClass       *inRear;      // pointer to the job record behind
}// class JobRecordClass
```

Figure 7.21

Java

```
class JobRecordClass
{
  public int                 jobNo;        // number of the job (integer)
  public JobRecordClass      inFront;      // reference to the job record in front
  public JobRecordClass      inRear;       // reference to the job record behind
} // class JobRecordClass
```

Figure 7.22

Data Encapsulation and Maintenance (contd)

- Only implementation details of `JobQueueClass` have changed

```
class JobQueueClass
{
  public:
    JobRecordClass      *frontOfQueue;     // pointer to the front of the queue
    JobRecordClass      *rearOfQueue;      // pointer to the rear of the queue

    void initializeJobQueue ( )
    {
      /*
       * initialize the job queue by setting frontOfQueue and rearOfQueue to NULL
       */
    }


    void addJobToQueue (int JobNumber)
    {
      /*
       * Create a new job record,
       * place jobNumber in its jobNo field,
       * set its inFront field to point to the current rearOfQueue
       * (thereby linking the new record to the rear of the queue),
       * and set its inRear field to NULL.
       * Set the inRear field of the record pointed to by the current rearOfQueue
       * to point to the new record (thereby setting up a two-way link), and
       * finally, set rearOfQueue to point to this new record.
       */
    }


    int removeJobFromQueue ( )
    {
      /*
       * set jobNumber equal to the jobNo field of the record at the front of the queue
       * update frontOfQueue to point to the next item in the queue,
       * set the inFront field of the record that is now the head of the queue to NULL,
       * and return jobNumber
       */
    }
}// class JobQueueClass
```

Figure 7.23

7.5 Abstract Data Types

- The problem with both implementations
  - There is only one queue, not three

- We need:
  - Data type + operations performed on instantiations of that data type

- Abstract data type

Abstract Data Type Example


```
class SchedulerClass
{
  . . .
  public:
    void queueHandler ( )
    {
      int               job1, job2;
      JobQueueClass     highPriorityQueue;
      JobQueueClass     mediumPriorityQueue;
      JobQueueClass     lowPriorityQueue;

        // some statements
      highPriorityQueue.initializeJobQueue ( );
        // some more statements
      mediumPriorityQueue.addJobToQueue (job1);
        // still more statements
      job2 = lowPriorityQueue.removeJobFromQueue ( );
        // even more statements
    }// queueHandler
  . . .
}// class SchedulerClass
```

Figure 7.24

- (Problems caused by `public` attributes solved later)

Another Abstract Data Type Example


```
class RationalClass
{
   public int      numerator;
   public int      denominator;

   public void sameDenominator (RationalClass r, RationalClass s)
   {
      // code to reduce r and s to the same denominator
   }

   public boolean equal (RationalClass t, RationalClass u)
   {
      RationalClass      v, w;
      v = t;
      w = u;
      sameDenominator (v, w);
      return (v.numerator == w.numerator);
   }

   // methods to add, subtract, multiply, and divide two rational numbers

}// class RationalClass
```

Figure 7.25

- (Problems caused by `public` attributes solved later)

7.6 Information Hiding

- Data abstraction
  - The designer thinks at the level of an ADT

- Procedural abstraction
  - Define a procedure — extend the language

- Both are instances of a more general design concept, *information hiding*
  - Design the modules in a way that items likely to change are hidden
  - Future change is localized
  - Changes cannot affect other modules

Information Hiding (contd)

- C++ abstract data type implementation with information hiding

```
class JobQueueClass
{
  // attributes
  private:
    int       queueLength;      // length of job queue
    int       queue[25];        // queue can contain up to 25 jobs

  // methods
  public:
    void initializeJobQueue ( )
    {
      // body of method unchanged from Figure 7.17
    }

    void addJobToQueue (int jobNumber)
    {
      // body of method unchanged from Figure 7.17
    }

    int removeJobFromQueue ( )
    {
      // body of method unchanged from Figure 7.17
    }
}// class JobQueueClass
```

Figure 7.26

Information Hiding (contd)

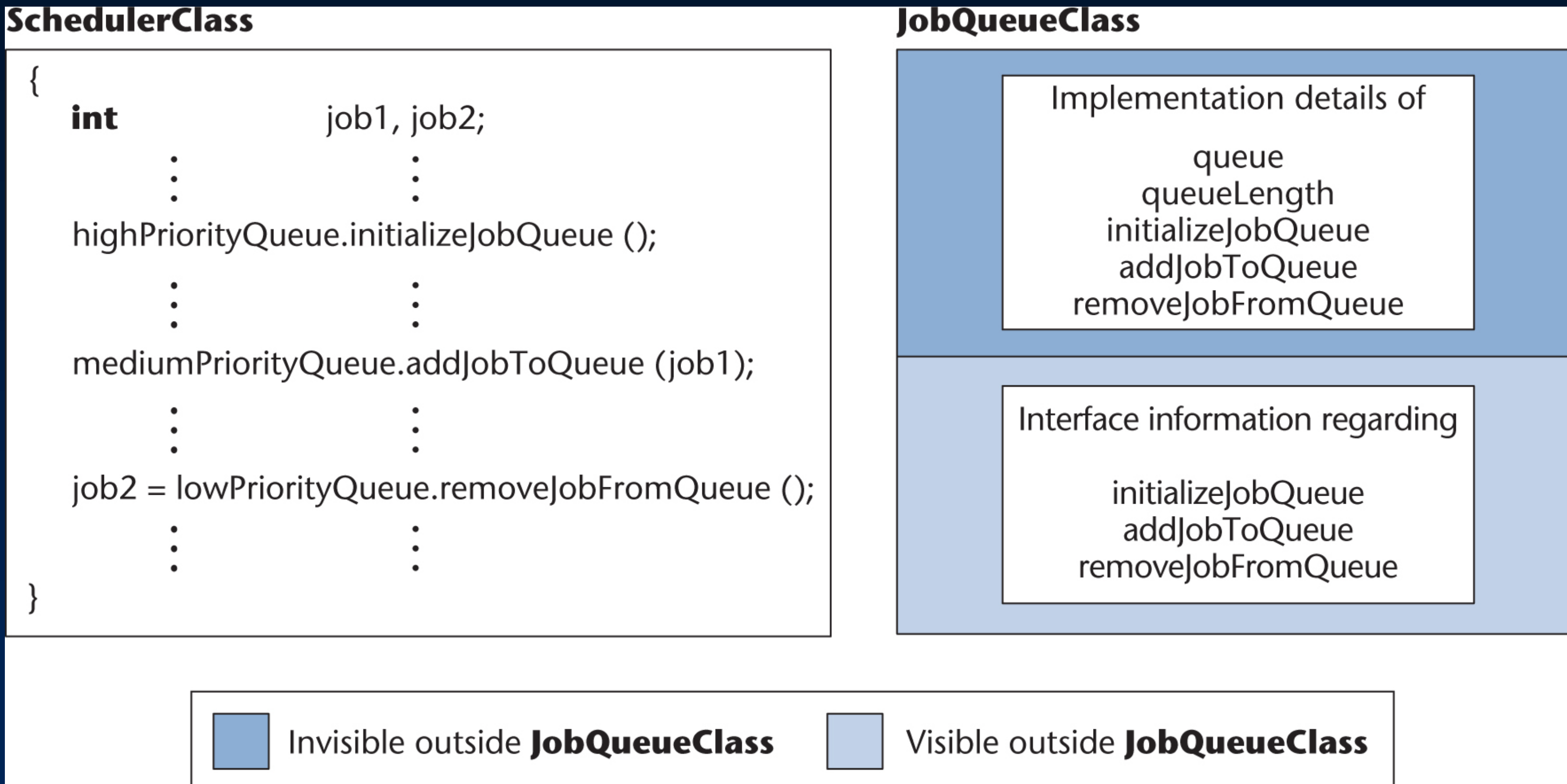

Figure 7.27

- Effect of information hiding via `private` attributes

Major Concepts of Chapter 7

$\Uparrow$

Objects (Section 7.7)

$\Uparrow$

Information hiding (Section 7.6)

$\Uparrow$

Abstract data types (Section 7.5)

$\Uparrow$

Data encapsulation (Section 7.4)

$\Uparrow$

Modules with high cohesion and low coupling (Sections 7.2 and 7.3)

$\Uparrow$

Modules (Section 7.1)

Figure 7.28

7.7 Objects

- First refinement
  - The product is designed in terms of abstract data types
  - Variables (“objects”) are instantiations of abstract data types

- Second refinement
  - Class: an abstract data type that supports *inheritance*
  - Objects are instantiations of classes

Inheritance

- Define **`HumanBeingClass`** to be a *class*
  - An instance of **`HumanBeingClass`** has *attributes*, such as
    - `age, height, gender`
  - Assign values to the attributes when describing an object

Inheritance (contd)

- Define **`ParentClass`** to be a *subclass* of **`HumanBeingClass`**
  - An instance of **`ParentClass`** has all the attributes of an instance of **`HumanBeingClass`**, plus attributes of his/her own
    - » `nameOfOldestChild, numberOfChildren`
  - An instance of **`ParentClass`** inherits all attributes of **`HumanBeingClass`**

Inheritance (contd)

- The property of inheritance is an essential feature of all object-oriented languages
  - Such as Smalltalk, C++, Ada 95, Java
- But not of classical languages
  - Such as C, COBOL or FORTRAN

Inheritance (contd)

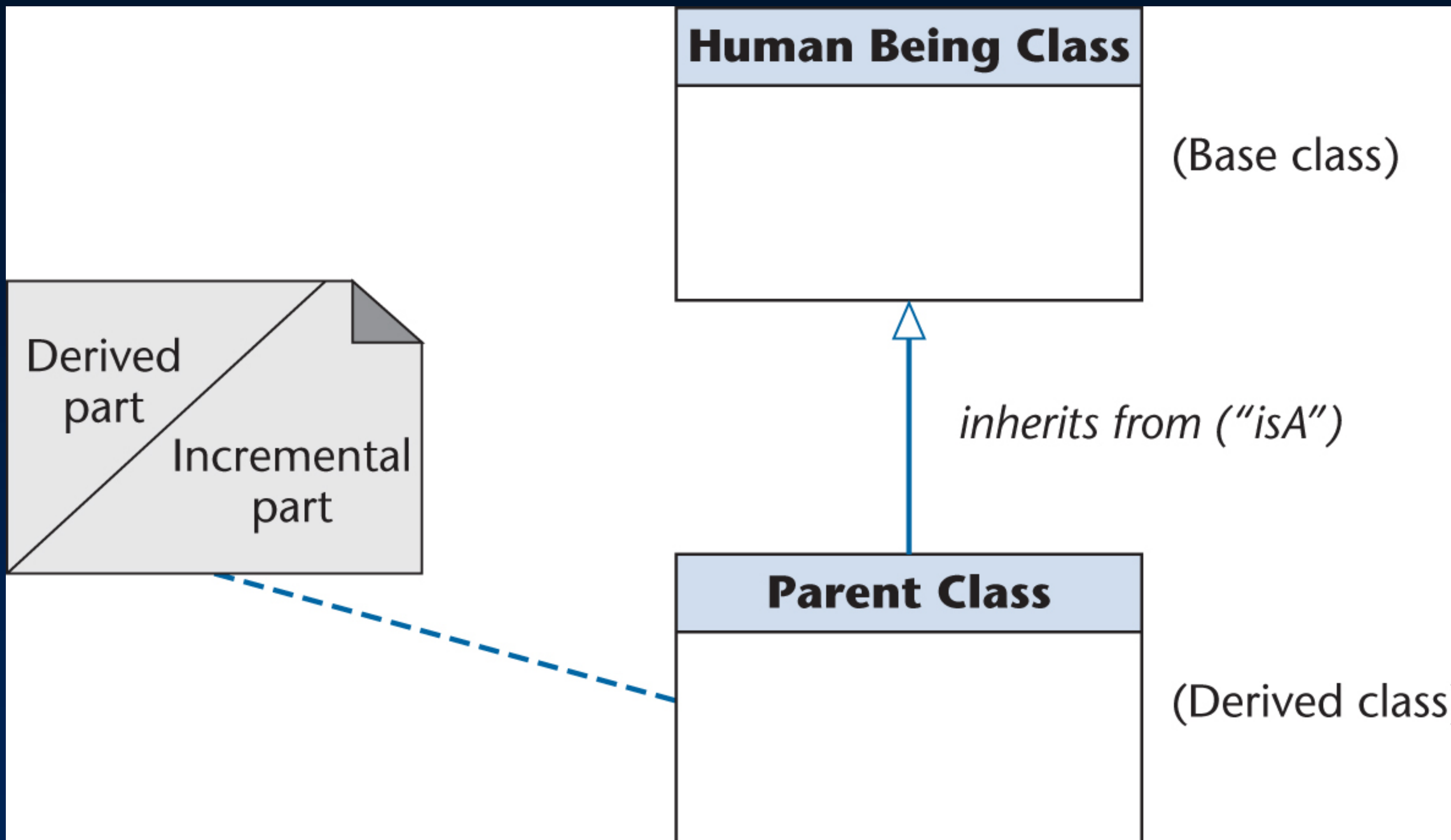

Figure 7.29

- UML notation
  - Inheritance is represented by a large open triangle

Java Implementation


```
class HumanBeingClass
{
   private int        age;
   private float      height;

   // public declarations of operations on HumanBeingClass

}// class HumanBeingClass


class ParentClass extends HumanBeingClass
{
   private String     nameOfOldestChild;
   private int        numberOfChildren;

   // public declarations of operations on ParentClass

}// class ParentClass
```

Figure 7.30

Aggregation

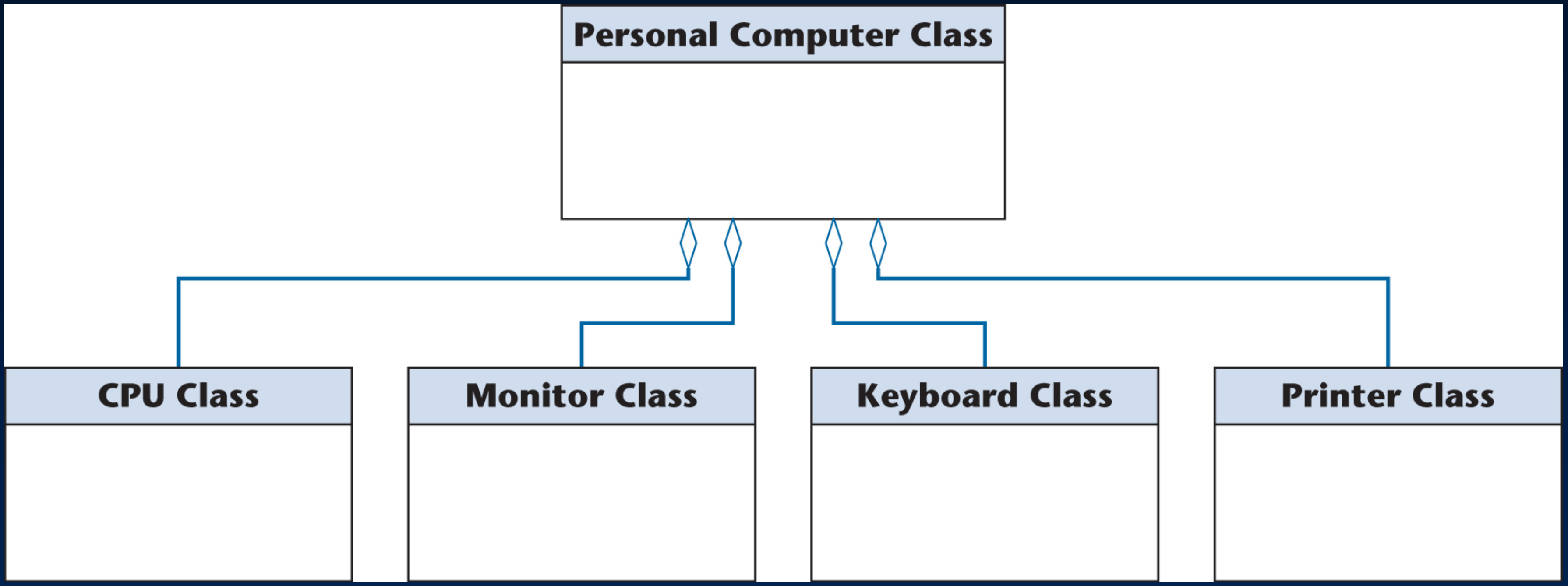

Figure 7.31

- UML notation for aggregation — open diamond

Association

Figure 7.32

- UML notation for association — line
  - Optional navigation triangle

Equivalence of Data and Action

- Classical paradigm
  - `record_1.field_2`

- Object-oriented paradigm
  - `thisObject.attributeB`
  - `thisObject.methodC ()`

7.8 Inheritance, Polymorphism and Dynamic Binding

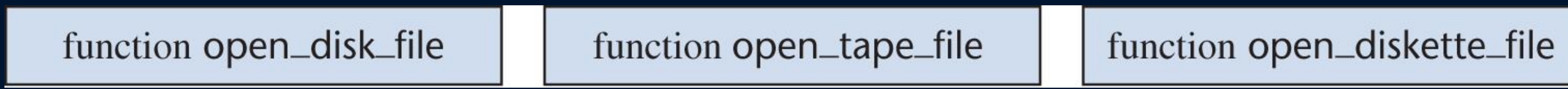

Figure 7.33a

- Classical paradigm
  - We must explicitly invoke the appropriate version

- Classical code to open a file
  - The correct method is explicitly selected

```
switch (file_type)
{
    case 1:
      open_disk_file ( );          // file_type 1 corresponds to a disk file
      break;
    case 2:
      open_tape_file ( );          // file_type 2 corresponds to a tape file
      break;
    case 3:
      open_diskette_file ( );      // file_type 3 corresponds to a diskette file
      break;
}
```

Figure 7.34(a)

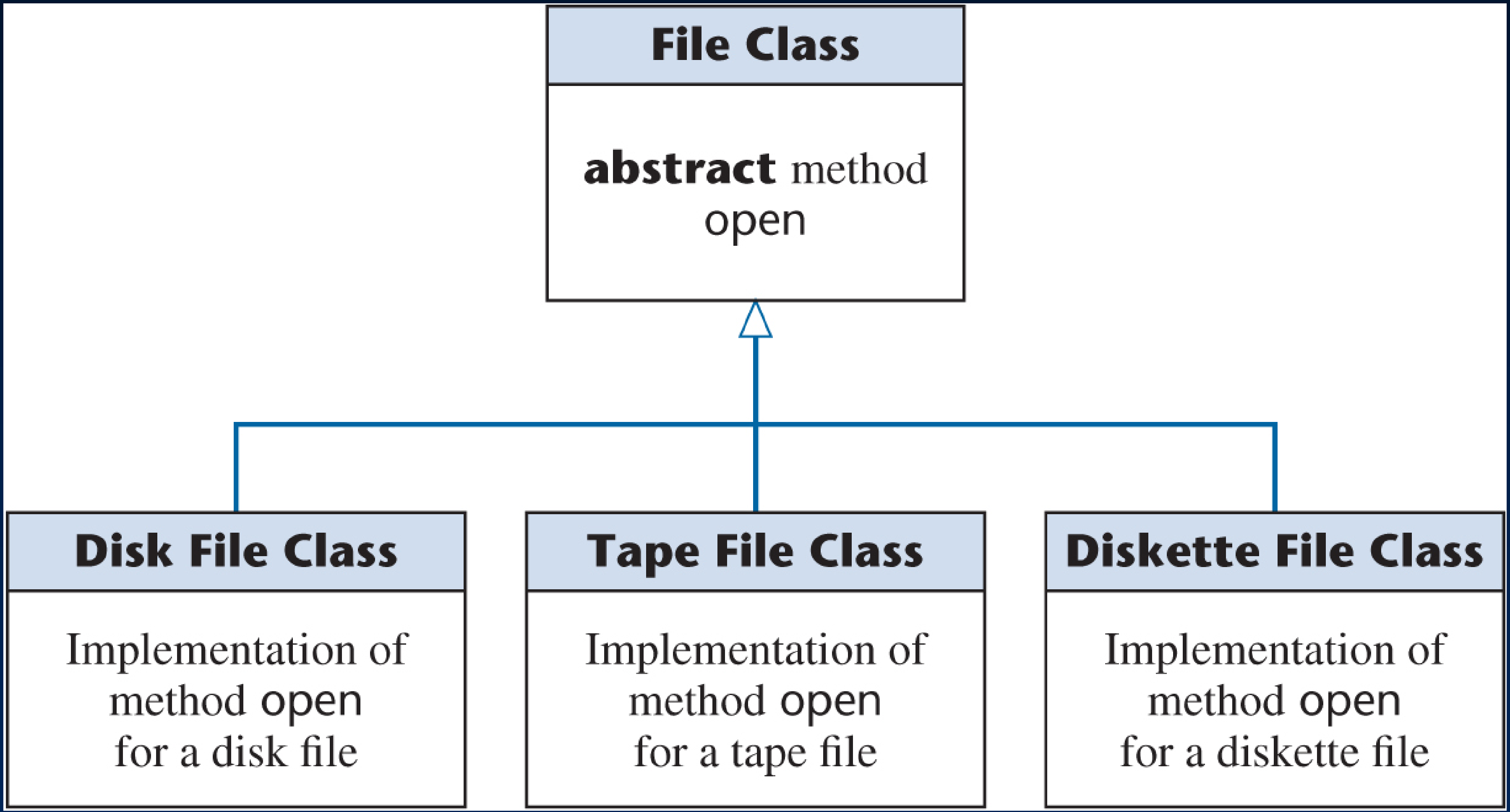

Figure 7.33(b)

- Object-oriented paradigm

Inheritance, Polymorphism and Dynamic Binding (contd)

- Object-oriented code to open a file
  - The correct method is invoked at run-time (dynamically)

```
myFile.open ( );
```

Figure 7.34(b)

- Method `open` can be applied to objects of different classes
  - “Polymorphic”

Inheritance, Polymorphism and Dynamic Binding (contd)

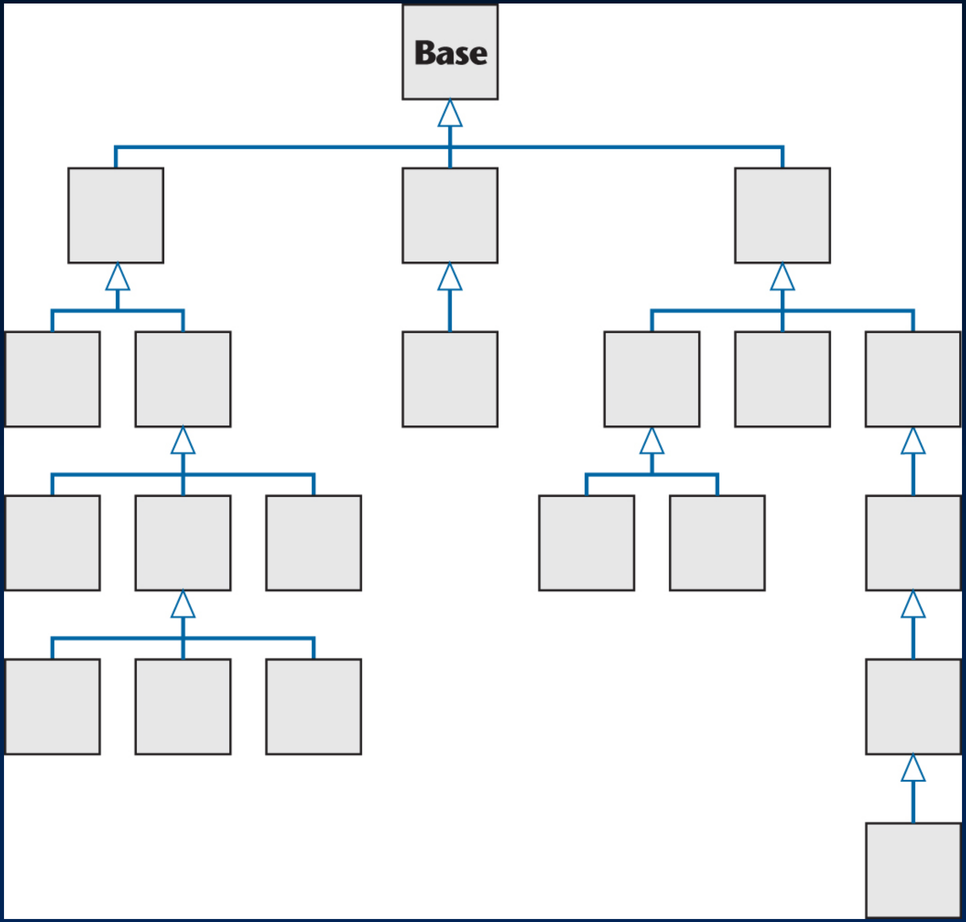

Figure 7.35

- `Method checkOrder (b : ` **`Base`**`)` can be applied to objects of any subclass of **`Base`**

- Polymorphism and dynamic binding
  - Can have a negative impact on maintenance
    - » The code is hard to understand if there are multiple possibilities for a specific method

- Polymorphism and dynamic binding
  - A strength and a weakness of the object-oriented paradigm

7.9 The Object-Oriented Paradigm

- Reasons for the success of the object-oriented paradigm
  - The object-oriented paradigm gives *overall* equal attention to data and operations
    - » At any one time, data or operations may be favored
  - A well-designed object (high cohesion, low coupling) models all the aspects of one physical entity
  - Implementation details are hidden

The Object-Oriented Paradigm (contd)

- The reason why the structured paradigm worked well at first
  - The alternative was no paradigm at all

The Object-Oriented Paradigm (contd)

- How do we know that the object-oriented paradigm is the best current alternative?
  - We don't
  - However, most reports are favorable
    - Experimental data (e.g., IBM [1994])
    - Survey of programmers (e.g., Johnson [2000])

Weaknesses of the Object-Oriented Paradigm

- Development effort and size can be large

- One's first object-oriented project can be larger than expected
  - Even taking the learning curve into account
  - Especially if there is a GUI

- However, some classes can frequently be reused in the next project
  - Especially if there is a GUI

Weaknesses of the Object-Oriented Paradigm (contd)

- Inheritance can cause problems
  - The fragile base class problem
  - To reduce the ripple effect, all classes need to be carefully designed up front
- Unless explicitly prevented, a subclass inherits all its parent's attributes
  - Objects lower in the tree can become large
  - "Use inheritance where appropriate"
  - Exclude unneeded inherited attributes

Weaknesses of the Object-Oriented Paradigm (contd)

- As already explained, the use of polymorphism and dynamic binding can lead to problems

- It is easy to write bad code in any language
  - It is especially easy to write bad object-oriented code

The Object-Oriented Paradigm (contd)

- Some day, the object-oriented paradigm will undoubtedly be replaced by something better
  - Aspect-oriented programming is one possibility
  - But there are many other possibilities