Chapter 14

## Design

**Learning Objectives**

After studying this chapter, you should be able to

* Perform the design workflow.
* Perform object-oriented design.
* Perform data flow analysis and transaction analysis.

Over the past 40 or so years, hundreds of design techniques have been put forward. Some are variations on existing techniques; others are radically different from anything previ- ously proposed. A few design techniques have been used by tens of thousands of software engineers; many have been used by only their authors. Some design strategies, particu- larly those developed by academics, have a firm theoretical basis. Others, including many drawn up by academics, are more pragmatic in nature; they were put forward because their authors found that they worked well in practice. Most design techniques are manual, but automation increasingly is becoming an important aspect of design, if only to assist in the management of documentation.

Notwithstanding this plethora of design techniques, a certain underlying pattern emerges. A major theme of this book is that two essential aspects of a product are its operations and the data on which the operations act. Therefore, the two basic ways of designing a product are operation-oriented design and data-oriented design. In **operation-oriented design**, the emphasis is on the operations. An example is data flow analysis (Section 14.3), where the objective is to design modules with high cohesion (Section 7.2). In **data-oriented design**, the data are considered first. For example, in Jackson’s technique (Section 14.5), the structure of the data is determined first, and then the procedures are designed to conform to the struc- ture of the data.

A weakness of operation-oriented design techniques is that they concentrate on the operations; the data are of only secondary importance. Data-oriented design techniques similarly emphasize the data, to the detriment of the operations. The solution is to use object-oriented techniques, which give equal weight to operations and data. In this chapter,

**465**

operation- and data-oriented design are described first, and then object-oriented design. Just as an object incorporates both operations and data, so object-oriented design combines features of operation-oriented and data-oriented design. Therefore, a basic understanding of operation- and data-oriented design is needed to get a full understanding of object- oriented design.

Before specific design techniques are examined, some general remarks must be made regarding design.

* 1. **Design and Abstraction: 14.1 Thiết kế và Trừu tượng hóa**

The classical design phase consists of three activities: architectural design, detailed design, and design testing. The input to the design process is the specification document, a descrip- tion of *what* the product is to do. The output is the design document, a description of *how* the product is to achieve this.

During **architectural design** (also known as **general design**, **logical design**, or **high-level design**), a modular decomposition of the product is developed. That is, the speci- fications are carefully analyzed, and a module structure that has the desired functionality is pro- duced. The output from this activity is a list of the modules and a description of how they are to be interconnected. From the viewpoint of abstraction, during architectural design, the existence of certain modules is assumed; the design then is developed in terms of those modules.

When the object-oriented paradigm is used, however, as explained in Section 1.9, the architectural design activity is performed during the object-oriented analysis workflow (Chapter 12). This is because the first step in the analysis workflow is to determine the classes. Because a class is a type of module, the modular decomposition has been per- formed during the analysis workflow.

The next activity in the classical design phase and a major activity of the object-oriented design workflow is **detailed design**, also known as **modular design**, **physical design**, or **low-level design**, during which each module (or class) is designed in detail. For example, specific algorithms are selected and data structures are chosen. Again, from the viewpoint of abstraction, during this activity the fact that the modules (or classes) are to be interconnected to form a complete product is ignored.

It was stated previously that the classical design phase has three activities and that the third activity is testing. The word *activity* was used, rather than *stage* or *step*, to emphasize that test- ing is an integral part of design, just as it is an integral part of the entire software development and maintenance process. Testing is not something performed only after the architectural design and detailed design have been completed. Similarly, in the case of object-oriented design, the test workflow is performed concurrently with the design workflow.

A variety of design techniques are now described, first operation-oriented techniques, then data-oriented techniques, and finally object-oriented techniques.

* 1. Operation-Oriented Design

Sections 7.2 and 7.3 made a theoretical case for decomposing a product into modules with high cohesion and low coupling. We now describe two practical classical techniques for achieving this design objective, data flow analysis (Section 14.3) and transaction analysis

(Section 14.4). In theory, data flow analysis can be applied whenever the specifications can be represented by a data flow diagram, and because (at least in theory) every product can be represented by a DFD, data flow analysis is universally applicable. In practice, however, in a number of situations, there are more appropriate design techniques, specifically for designing products where the flow of data is secondary to other considerations. Examples where other design techniques are indicated include rule-based systems (expert systems), databases, and transaction-processing products. (Transaction analysis, described in Section 14.4, is a good way of decomposing transaction-processing products into modules.

(Dihcj: 14.2 Thiết kế định hướng hoạt động

Phần 7.2 và 7.3 đưa ra trường hợp lý thuyết để phân tách sản phẩm thành các mô-đun có tính liên kết cao và tính liên kết thấp. Bây giờ chúng tôi mô tả hai kỹ thuật cổ điển thực tế để đạt được mục tiêu thiết kế này, phân tích luồng dữ liệu (Phần 14.3) và phân tích giao dịch

(Mục 14.4). Về lý thuyết, phân tích luồng dữ liệu có thể được áp dụng bất cứ khi nào các thông số kỹ thuật có thể được biểu thị bằng sơ đồ luồng dữ liệu và bởi vì (ít nhất là trên lý thuyết) mọi sản phẩm đều có thể được biểu thị bằng DFD, nên phân tích luồng dữ liệu được áp dụng phổ biến. Tuy nhiên, trên thực tế, trong một số trường hợp, có nhiều kỹ thuật thiết kế phù hợp hơn, đặc biệt là để thiết kế các sản phẩm mà luồng dữ liệu là thứ yếu so với các cân nhắc khác. Các ví dụ về các kỹ thuật thiết kế khác được chỉ ra bao gồm các hệ thống dựa trên quy tắc (hệ thống chuyên gia), cơ sở dữ liệu và các sản phẩm xử lý giao dịch. (Phân tích giao dịch, được mô tả trong Phần 14.4, là một cách tốt để phân tách các sản phẩm xử lý giao dịch thành các mô-đun.))

* 1. Data Flow Analysis

**Data flow analysis (DFA)** is a classical design technique for achieving modules with high cohesion. It can be used in conjunction with most analysis techniques. Here, DFA is presented in conjunction with structured systems analysis (Section 12.3). The input to the technique is a data flow diagram. A key point is that, once the DFD has been completed, the software designer has precise and complete information regarding the input to and output from the product.

Consider the flow of data in the product represented by the DFD of Figure 14.1. The product somehow transforms input into output. At some point in the DFD, the input ceases to be input and becomes some sort of internal data. Then, at some further point, these internal data take on the quality of output. This is shown in more detail in Figure 14.2. The point at which the input loses the quality of being input and simply becomes internal data operated on by the product is termed the **point of highest abstraction of input**. The **point of highest abstraction of output** is similarly the first point in the flow of data at which the output can be identified as such, rather than as some sort of internal data.

Using the points of highest abstraction of input and output, the product is decomposed into three modules: input\_module, transform\_module, and output\_module. Now each mod- ule is taken in turn, its points of highest abstraction found, and the module decomposed again. This procedure is continued stepwise until each module performs a single operation; that is, the

**FIGURE 14.1** A data flow diagram showing flow of data and operations of product.

Input

a

b

c

d

e

f

g

h

Output

**FIGURE 14.2** Points of highest abstraction of input and output.

Input

a

b

c

d

e

f

g

h

Output

input\_module

transform\_module

output\_module

Point of highest abstraction

of input

Point of highest abstraction

of output

design consists of modules with high cohesion. Consequently, stepwise refinement, the founda- tion of so many other software engineering techniques, also underlies data flow analysis.

In fairness, it should be pointed out that minor modifications might have to be made to the decomposition to achieve the lowest possible coupling. Data flow analysis is a way of achieving high cohesion. The aim of composite/structured design is high cohesion but also low coupling. To achieve the latter, sometimes it is necessary to make minor modifications to the design. For example, because DFA does not take coupling into account, control coupling may arise inadvertently in a design constructed using DFA. In such a case, all that is needed is to modify the two modules involved so that data, and not control, are passed between them.

*MiniCase Study*

*Mini Case Study Word Counting*



14.3.1

Consider the problem of designing a product that takes as input a file name and returns the number of words in that file, similarly to the UNIX wc utility.

Figure 14.3 depicts the data flow diagram. There are five modules. Module read\_file\_name reads the name of the file, which then is validated by validate\_file\_ name. The validated name is passed to count\_number\_of\_words, which does pre- cisely that. The word count is passed on to format\_word\_count, and the formatted word count finally is passed to display\_word\_count for output.

Examining the data flow, the initial input is file\_name. When this becomes vali- dated\_file\_name, it still is a file name and therefore has not lost its quality of being input data. But consider module count\_number\_of\_words. Its input is validated\_ file\_name, and its output is word\_count. The output from this module is totally different in quality from the input to the product as a whole. It is clear that the point of highest abstraction of input is as indicated on Figure 14.3. Similarly, even though the output from count\_number\_of\_words undergoes some sort of formatting, it is essentially *output* from the time it emerges from module count\_number\_of\_words. The point of highest abstraction of output therefore is as shown in Figure 14.3.

The result of decomposing the product using these two points of highest abstrac- tion is shown in the structure chart of Figure 14.4. This figure also reveals that the data

**FIGURE 14.3** The first refinement of the data flow diagram.

file\_ name

read\_ file\_ name

file\_ name

validate\_ file\_name file\_

name

validated\_

word\_ count\_ count number\_

of\_words

format\_ word\_ count

formatted\_ word\_

count

display\_ word\_ count

desired\_ output

Input to here

Point of highest abstraction

of input

Output from here

Point of highest abstraction

of output

**FIGURE 14.4**

The first refinement of the structure chart.

validated\_ file\_name

status\_flag

valid file\_



word\_count

\_ t



|  |  |
| --- | --- |
| perform\_ word\_ count | |
| ated\_ name | word coun |
| count\_ number\_of\_ words | |

read\_and\_ validate\_ file\_name

format\_ and\_display\_ word\_count

Data Control

**FIGURE 14.5** The second refinement of the structure chart.



|  |  |
| --- | --- |
| perform\_ word\_ count | |
| ated\_ name | word coun |
| count\_ number\_of\_ words | |

validated\_ file\_name

status\_flag

valid file\_



\_ word\_count t

get\_ input

produce\_ output

file\_name



file\_name status\_flag

word\_count

formatted\_ word\_count

formatted\_ word\_count



read\_ file\_ name

validate\_ file\_ name

format\_ word\_ count

display\_ word\_ count

Data Control

flow diagram of Figure 14.3 is somewhat too simplistic. The DFD does not show the logical flow corresponding to what happens if the file specified by the user does not exist. Module read\_and\_validate\_file\_name must return a status\_flag to perform\_ word\_count. If the name is invalid, then it is ignored by perform\_word\_count and an error message of some sort is printed. But, if the name is valid, it is passed on to count\_number\_of\_words. In general, wherever there is a conditional data flow, a corresponding control flow is needed.

As explained in Section 7.2.5, a module has communicational cohesion if it per- forms a series of operations related by the sequence of steps to be followed by the product and if all the operations are performed on the same data. In Figure 14.4, two modules have communicational cohesion: read\_and\_validate\_file\_name and format\_and\_display\_word\_count. These must be decomposed further. The final result is shown in Figure 14.5. All eight modules have functional cohesion, with either data coupling (Section 7.3.5) or no coupling between them.

**FIGURE 14.6**

Module name Module type Return type Input arguments

Output arguments Error messages Files accessed

Files changed

Modules called Narrative

**read\_file\_name**

Function **string** None None None None None None

The product is invoked by the user by means of the command string

**word\_count** <**file\_name**>

Using an operating system call, this module accesses the contents of the command string input by the user, extracts <**file\_name**>**,** and returns it as the value of the module.

The detailed design of four modules of the example.

Now that the architectural design has been completed, the next step is the detailed design. Here, data structures are chosen and algorithms selected. The detailed design of each module then is handed to a programmer for implementation. Just as with virtually every other phase of software production, time constraints usually require that the implementation be done by a team, rather than having a single programmer responsible for coding all the modules. For this reason, the detailed design of each module must be presented so it can be understood without reference to any other module. The detailed design of four of the eight modules appears in Figure 14.6; the other four modules are presented in a different format.



Module name Module type Return type Input arguments

Output arguments Error messages Files accessed

Files changed Modules called

Narrative

**validate\_file\_name**

Function

**Boolean file\_name : string** None

None None None None

This module makes an operating system call to determine whether file **file\_name** exists. The module returns **true** if the file exists and **false** otherwise.

**FIGURE 14.6**

(*continued*)

Module name Module type Return type Input arguments

Output arguments Error messages Files accessed

Files changed Modules called

Narrative

**count\_number\_of\_words**

Function

**integer validated\_file\_name : string** None

None None None None

This module determines whether **validated\_file\_name** is a text file, that is, divided into lines of characters. If so, the module returns the number of words in the text file;

otherwise, the module returns —**1**.

Module name Module type Return type Input arguments

Output arguments Error messages Files accessed

Files changed Modules called

Narrative

**produce\_output**

Function

**void**

**word\_count : integer**

None None None None

**format\_word\_count**

arguments: **word\_count : integer**

**formatted\_word\_count : string display\_word\_count**

arguments: **formatted\_word\_count : string**

This module takes the integer **word\_count** passed to it by the calling module and calls **format\_word\_count** to have that integer formatted according to the specifications. Then it calls **display\_word\_count** to have the line printed.

The design of Figure 14.6 is independent of the programming language. How- ever, if management decides on an implementation language before the detailed design is started, the use of a **program description language (PDL)** for representing the detailed design is an attractive alternative (**pseudocode** is an earlier name for PDL). PDL essentially consists of comments connected by the control statements of the chosen implementation language. Figure 14.7 shows a

**FIGURE 14.7**

PDL

(pseudocode) representation of the detailed design of four methods of the example.

**void** perform\_word\_count ( )

{

String validated\_file\_name;

**Int** word\_count;

**if** (get\_input (validated\_file\_name) *is* **null**)

*print* “error 1: file does not exist”;

**else**

{

set word\_count *equal to* count\_number\_of\_words (validated\_file\_name);

**if** (word\_count *is equal to* –1)

*print* “error 2: file is not a text file”;

**else**

produce\_output (word\_count);

}

}

String get\_input ( )

{

String file\_name;

file\_name = read\_file\_name ( );

**if** (validate\_file\_name (file\_name) *is* **true**)

{

**return** file\_name;

}

**else**

**return null**;

}

**void** display\_word\_count (String formatted\_word\_count)

{

*print* formatted\_word\_count, *left justified*;

}

String format\_word\_count (**int** word\_count);

{

**return** “File contains” word\_count “words”;

}

detailed design for the remaining four modules of the product written in a PDL with the flavor of C++ or Java. A PDL has the advantage that it generally is clear and concise, and the implementation step usually consists merely of translating the comments into the relevant programming language. The weakness is that sometimes there is a tendency for the designers to go into too much detail and produce a complete code implementation of a module rather than a PDL detailed design.

After it has been fully documented and successfully tested, the detailed design is handed over to the implementation team for coding. The product then proceeds through the remaining phases of the classical software life cycle.



* **Iterate**

Find the point of highest abstraction of input of each input stream. Find the point of highest abstraction of output of each output stream.

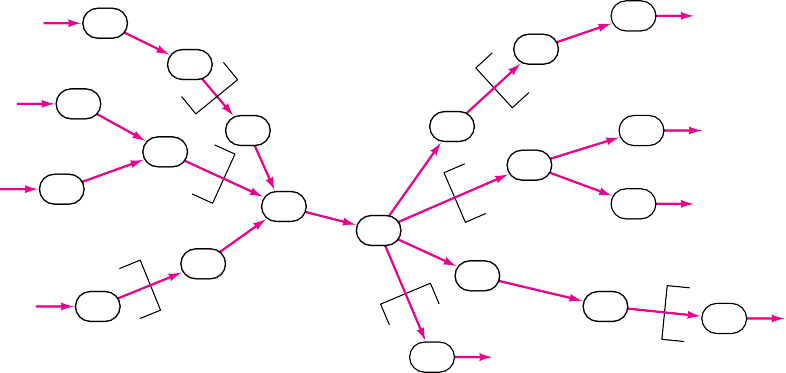
Decompose the data flow diagram using these points of highest abstraction.

* **Until** the resulting modules have high cohesion.
* If a resulting coupling is too high, adjust the design.

**Box 14.1**

**How to Perform Data Flow Analysis**

**FIGURE 14.8**



i1

o1

i2

o2

o3

i4

o5

The data flow diagram with multiple input and output streams.

i3

o4

14.3.2 Data Flow Analysis Extensions

The reader may well feel that this mini case study is somewhat artificial, in that the data flow diagram (Figure 14.3) has only one input stream and one output stream. To see what happens in more complex situations, consider Figure 14.8. Now there are four input streams and five output streams, a situation that corresponds more closely to reality.

When there are multiple input and output streams, the way to proceed is to find the point of highest abstraction of input for each input stream and the point of highest abstraction of output for each output stream. Use these points to decompose the given data flow diagram into modules with fewer input–output streams than the original. Continue this way until each resulting module has high cohesion. Finally, determine the coupling between each pair of modules and make any necessary adjustments.

Data flow analysis is summarized in How to Perform Box 14.1.

* 1. Transaction Analysis

A **transaction** is an operation from the viewpoint of the user of the product, such as “pro- cess a request” or “print a list of today’s orders.” Data flow analysis is inappropriate for the transaction-processing type of product, in which a number of related operations, similar in outline but differing in detail, must be performed. A typical example is the software controlling

raw\_

trans\_t4

raw\_

trans\_t5

audit\_

information

audit\_

information

trans\_t2



* Design the architecture in terms of two components: The analyzer.

The dispatcher.

* **For** each set of related operations

Design one basic module and instantiate it as many times as necessary.

**Box 14.2**

**How to Perform Transaction Analysis**

an automated teller machine. The customer inserts a card with a magnetic strip into a slot, keys in a password, and then performs operations such as deposit to a checking, savings, or credit card account; withdraw from an account; or determine the balance in an account. This type of product is depicted in Figure 14.9. A good way to design such a product is to break it into two pieces, the analyzer and the dispatcher. The analyzer determines the transaction type and passes this information to the dispatcher, which performs the transaction.

As explained in Section 7.2.2, a module has logical cohesion when it performs a series of related operations, one of which is selected by the calling module. The design shown in Figure 14.10 is undesirable, because it has two modules with logical cohesion (Section 7.2.2), edit\_any\_transaction and update\_any\_file. On the other hand, it seems a waste of effort to have five very similar edit modules and five very similar update modules. The

**FIGURE 14.9** A typical transaction-processing system.



edit\_ transaction\_ t1

good\_ trans\_t1

update\_ file\_v

edit\_ transaction\_ t2

good\_ trans\_t2

update\_ file\_w

raw\_ trans

determine\_ transaction\_ type

raw\_ trans\_t3

edit\_ transaction\_ t3

good\_ trans\_t3

update\_ audit\_

file\_x information

write\_to\_ audit\_trail

edit\_ transaction\_ t4

good\_ trans\_t4

update\_ file\_y

edit\_ transaction\_ t5

good\_ trans\_t5

update\_ file\_z

audit\_

information

audit\_

information

raw\_

trans\_t1

raw\_

**FIGURE 14.10**



process\_ transaction

analyzer

dispatcher

process\_ trans\_t1

process\_ trans\_t2

process\_ trans\_t3

process\_ trans\_t4

process\_ trans\_t5

edit\_any\_ transaction

update\_ any\_file

write\_to\_ audit\_trail

A poor design of transaction- processing system.

solution is software reuse (Section 8.1): A basic edit module should be designed, coded, documented, tested, and then instantiated five times. Each version is slightly different, but the differences are small enough to make this approach worthwhile. Similarly, a basic update module can be instantiated five times and slightly modified to cater to the five dif- ferent update types. The resulting design has high cohesion and low coupling.

**Transaction analysis** is summarized in How to Perform Box 14.2.

* 1. Data-Oriented Design : 14.5 Thiết kế hướng dữ liệu

The basic principle behind data-oriented design is to design the product according to the structure of the data on which it is to operate. That is, first the structure of the data is determined. Then each procedure is given the same structure as the data on which it oper- ates. There are a number of data-oriented techniques of this type; the most well known are those of Michael Jackson [1975], Warnier [1976], and Orr [1981]. The three techniques share many similarities.

Data-oriented design was never as popular as operation-oriented design and, with the rise of the object-oriented paradigm, it has largely fallen out of fashion. For reasons of space, data-oriented design is not discussed further in this book; the interested reader should consult the references cited in the previous paragraph.

(Dịch: Nguyên tắc cơ bản đằng sau thiết kế hướng dữ liệu là thiết kế sản phẩm theo cấu trúc của dữ liệu mà nó hoạt động. Đó là, đầu tiên cấu trúc của dữ liệu được xác định. Sau đó, mỗi thủ tục được đưa ra cấu trúc giống như dữ liệu mà nó hoạt động. Có một số kỹ thuật hướng dữ liệu thuộc loại này; nổi tiếng nhất là của Michael Jackson [1975], Warnier [1976] và Orr [1981]. Ba kỹ thuật chia sẻ nhiều điểm tương đồng.

Thiết kế hướng dữ liệu chưa bao giờ phổ biến như thiết kế hướng hoạt động và cùng với sự gia tăng của mô hình hướng đối tượng, nó phần lớn đã lỗi thời. Vì lý do không gian, thiết kế hướng dữ liệu không được thảo luận thêm trong cuốn sách này; người đọc quan tâm nên tham khảo các tài liệu tham khảo được trích dẫn trong đoạn trước.)

* 1. Object-Oriented Design : 14.6 Thiết kế hướng đối tượng

As previously stated, the Unified Process assumes previous knowledge of **object- oriented design (OOD)**. Accordingly, we now describe OOD and then discuss the de- sign workflow of the Unified Process in Section 14.9.

The aim of OOD is to design the product in terms of objects, that is, instantiations of the classes and subclasses extracted during object-oriented analysis. Classical languages, such as C, and older (pre-2000) versions of COBOL and Fortran do not support objects as such. This might seem to imply that OOD is accessible only to users of object-oriented languages like Smalltalk [Goldberg and Robson, 1989], C++ [Stroustrup, 2003], Ada 95 [ISO/IEC 8652, 1995], and Java [Flanagan, 2005].

That is not the case. Although OOD as such is not supported by classical languages, a large subset of OOD can be used. As explained in Section 7.7, a class is an abstract data type with inheritance and an object is an instance of a class. When using an implementa- tion language that does not support inheritance, the solution is to utilize those aspects of OOD that can be achieved in the programming language used in the project, that is, to use **abstract data type design**. Abstract data types can be implemented in virtually any language that supports **type** statements. Even in a classical language that does not support type statements as such, and hence cannot support abstract data types, it still may be possible to implement data encapsulation. Figure 7.28 depicts a hierarchy of design concepts starting with modules and ending with objects. In those cases where full OOD is not possible, the developers should endeavor to ensure that their design uses the high- est possible concept in the hierarchy of Figure 7.28 that their implementation language supports.

The two key steps of OOD are to complete the class diagram and perform the detailed design. With regard to the first step, completing the **class diagram**, the formats of the attributes need to be determined, and the methods need to be assigned to the relevant classes. The formats of the attributes can generally be deduced directly from the analysis artifacts. For example, in the United States the specifications may state that a date such as December 3, 1947, shall be represented as 12/03/1947 (mm/dd/yyyy format) or in Europe as 03/12/1947 (dd/mm/yyyy format). But, irrespective of which date conven- tion is used, a total of 10 characters is needed.

The information for determining the formats is obtained during the analysis work- flow, so the formats could certainly be added to the class diagram at that time. However, the object-oriented paradigm is iterative. Each iteration results in a change to what has already been completed. For practical reasons, then, information should be added to UML models as late as possible. Consider, for example, Figures 13.21 and 13.22, which show the first two iterations of the class diagram of the MSG Foundation case study. Neither of those two iterations shows the attributes of the classes. If the attributes had been determined earlier, they would probably have had to be modified, as well as possibly

moved from class to class, until the analysis team was satisfied with the class diagram. Instead, all that had to be modified was the classes themselves. In general, it makes little sense to add an item to a class diagram (or any other UML diagram) before it is abso- lutely essential to do so, because adding the item will make the next iteration unneces- sarily burdensome. In particular, it makes little sense to specify formats before they are strictly needed.

The other major component of the first step of OOD is to assign methods (implementa- tions of operations) to classes. Determination of all the operations of the product is per- formed by examining the interaction diagrams of every scenario. This is straightforward. The hard part is to determine how to decide which methods should be associated with each class.

A method can be assigned either to a class or to a client that sends a message to an object of that class. (A client of an object is a program unit that sends a message to that object.) One principle that can be employed to assist in deciding how to assign an operation is information hiding (Section 7.6). That is, the state variables of a class should be declared **private** (accessible only within an object of that class) or **protected** (accessible only within an object of that class or a subclass of that class). Accordingly, operations performed on state variables must be local to that class.

A second principle is that, if a particular operation is invoked by a number of different clients of an object, it makes sense to have a single copy of that operation implemented as a method of the object, rather than have a copy in each client of that object.

A third principle that can be employed to assist in deciding where to locate a method is to use responsibility-driven design. As explained in Section 1.9, **responsibility-driven design** is a key aspect of the object-oriented paradigm. If a client sends a message to an object, then that object is responsible for every aspect of carrying out the request of the client. The client does not know how the request will be carried out and is not permitted to know. Once the request has been carried out, control returns to the client. At that point, all the client knows is that the request has been carried out; it still has no idea how this was achieved.

To see how these principles are utilized, we now illustrate OOD by means of two examples. As before, the elevator problem case study is presented, with just one eleva- tor for simplicity. Then, we return to the MSG Foundation case study. By using the same examples, you can compare different approaches without having to worry about the rami- fications of the problem itself.

(Dịch: Như đã nêu trước đây, Quy trình hợp nhất giả định kiến ​​thức trước đó về thiết kế hướng đối tượng (OOD). Theo đó, bây giờ chúng ta mô tả OOD và sau đó thảo luận về quy trình thiết kế của Quy trình hợp nhất trong Phần 14.9.

Mục đích của OOD là thiết kế sản phẩm dưới dạng các đối tượng, tức là các thể hiện của các lớp và các lớp con được trích xuất trong quá trình phân tích hướng đối tượng. Các ngôn ngữ cổ điển, chẳng hạn như C và các phiên bản cũ hơn (trước năm 2000) của COBOL và Fortran không hỗ trợ các đối tượng như vậy. Điều này dường như ngụ ý rằng OOD chỉ có thể truy cập được đối với người dùng các ngôn ngữ hướng đối tượng như Smalltalk [Goldberg và Robson, 1989], C++ [Stroustrup, 2003], Ada 95 [ISO/IEC 8652, 1995] và Java [Flanagan, 2005].

Đó không phải là tình huống. Mặc dù OOD như vậy không được các ngôn ngữ cổ điển hỗ trợ, nhưng một tập hợp con lớn của OOD có thể được sử dụng. Như đã giải thích trong Phần 7.7, một lớp là một kiểu dữ liệu trừu tượng với tính kế thừa và một đối tượng là một thể hiện của một lớp. Khi sử dụng ngôn ngữ triển khai không hỗ trợ tính kế thừa, giải pháp là tận dụng các khía cạnh của OOD có thể đạt được bằng ngôn ngữ lập trình được sử dụng trong dự án, tức là sử dụng thiết kế kiểu dữ liệu trừu tượng. Các kiểu dữ liệu trừu tượng có thể được triển khai trong hầu hết mọi ngôn ngữ hỗ trợ các câu lệnh kiểu. Ngay cả trong một ngôn ngữ cổ điển không hỗ trợ các câu lệnh kiểu như vậy và do đó không thể hỗ trợ các kiểu dữ liệu trừu tượng, vẫn có thể thực hiện đóng gói dữ liệu. Hình 7.28 mô tả một hệ thống phân cấp các khái niệm thiết kế bắt đầu bằng các mô-đun và kết thúc bằng các đối tượng. Trong những trường hợp không thể thực hiện được toàn bộ OOD, các nhà phát triển nên cố gắng đảm bảo rằng thiết kế của họ sử dụng khái niệm cao nhất có thể trong hệ thống phân cấp của Hình 7.28 mà ngôn ngữ triển khai của họ hỗ trợ.

Hai bước chính của OOD là hoàn thành sơ đồ lớp và thực hiện thiết kế chi tiết. Đối với bước đầu tiên, hoàn thiện biểu đồ lớp, cần xác định dạng thức của các thuộc tính và các phương thức cần được gán cho các lớp liên quan. Các định dạng của các thuộc tính nói chung có thể được suy ra trực tiếp từ các tạo phẩm phân tích. Ví dụ: ở Hoa Kỳ, các thông số kỹ thuật có thể nêu rõ rằng một ngày như ngày 3 tháng 12 năm 1947 sẽ được thể hiện là 03/12/1947 (định dạng mm/dd/yyyy) hoặc ở Châu Âu là 12/03/1947 (dd/ mm/yyyy). Tuy nhiên, bất kể quy ước ngày nào được sử dụng, tổng cộng 10 ký tự là cần thiết.

Thông tin để xác định các định dạng thu được trong quá trình phân tích, vì vậy các định dạng chắc chắn có thể được thêm vào sơ đồ lớp tại thời điểm đó. Tuy nhiên, mô hình hướng đối tượng là lặp đi lặp lại. Mỗi lần lặp dẫn đến một thay đổi đối với những gì đã được hoàn thành. Do đó, vì những lý do thực tế, thông tin nên được thêm vào các mô hình UML càng muộn càng tốt. Ví dụ, xem xét Hình 13.21 và 13.22, cho thấy hai lần lặp lại đầu tiên của sơ đồ lớp trong nghiên cứu tình huống của Tổ chức MSG. Cả hai lần lặp đó đều không hiển thị các thuộc tính của các lớp. Nếu các thuộc tính đã được xác định sớm hơn, chúng có lẽ đã phải được sửa đổi, cũng như có thể chuyển từ lớp này sang lớp khác, cho đến khi nhóm phân tích hài lòng với sơ đồ lớp. Thay vào đó, tất cả những gì phải sửa đổi là bản thân các lớp. Nói chung, việc thêm một mục vào sơ đồ lớp (hoặc bất kỳ sơ đồ UML nào khác) trước khi thực sự cần thiết phải làm như vậy là vô nghĩa, bởi vì việc thêm mục đó sẽ khiến lần lặp tiếp theo trở nên nặng nề một cách không cần thiết. Đặc biệt, sẽ rất khó để chỉ định các định dạng trước khi chúng thực sự cần thiết.

Thành phần chính khác của bước đầu tiên của OOD là gán các phương thức (việc triển khai các hoạt động) cho các lớp. Việc xác định tất cả các hoạt động của sản phẩm được thực hiện bằng cách kiểm tra các sơ đồ tương tác của mọi tình huống. Điều này là đơn giản. Phần khó là xác định cách quyết định phương thức nào sẽ được liên kết với mỗi lớp.

Một phương thức có thể được gán cho một lớp hoặc cho một máy khách gửi thông báo đến một đối tượng của lớp đó. (Máy khách của một đối tượng là một đơn vị chương trình gửi thông báo đến đối tượng đó.) Một nguyên tắc có thể được sử dụng để hỗ trợ quyết định cách chỉ định một thao tác là ẩn thông tin (Phần 7.6). Nghĩa là, các biến trạng thái của một lớp phải được khai báo là private (chỉ có thể truy cập trong một đối tượng của lớp đó) hoặc được bảo vệ (chỉ có thể truy cập trong một đối tượng của lớp đó hoặc một lớp con của lớp đó). Theo đó, các hoạt động được thực hiện trên các biến trạng thái phải là cục bộ của lớp đó.

Nguyên tắc thứ hai là, nếu một thao tác cụ thể được gọi bởi một số máy khách khác nhau của một đối tượng, thì nên có một bản sao duy nhất của thao tác đó được triển khai như một phương thức của đối tượng, thay vì có một bản sao trong mỗi máy khách của đối tượng. đối tượng đó.

Nguyên tắc thứ ba có thể được sử dụng để hỗ trợ quyết định nơi xác định phương pháp là sử dụng thiết kế hướng đến trách nhiệm. Như đã giải thích trong Phần 1.9, thiết kế hướng đến trách nhiệm là một khía cạnh quan trọng của mô hình hướng đối tượng. Nếu một máy khách gửi một thông báo đến một đối tượng, thì đối tượng đó chịu trách nhiệm về mọi khía cạnh của việc thực hiện yêu cầu của máy khách. Khách hàng không biết yêu cầu sẽ được thực hiện như thế nào và không được phép biết. Khi yêu cầu đã được thực hiện, quyền kiểm soát sẽ trở lại máy khách. Tại thời điểm đó, tất cả những gì khách hàng biết là yêu cầu đã được thực hiện; nó vẫn không biết làm thế nào đạt được điều này.

Để xem các nguyên tắc này được sử dụng như thế nào, bây giờ chúng tôi minh họa OOD bằng hai ví dụ. Như trước đây, nghiên cứu điển hình về vấn đề thang máy được trình bày, chỉ với một thang máy để đơn giản hóa. Sau đó, chúng tôi trở lại nghiên cứu tình huống của Quỹ MSG. Bằng cách sử dụng các ví dụ giống nhau, bạn có thể so sánh các cách tiếp cận khác nhau mà không phải lo lắng về sự phân nhánh của chính vấn đề.)

*Case Study*



* 1. Object-Oriented Design: The Elevator Problem Case Study

**Step 1. Complete the Class Diagram**

A design workflow detailed class diagram (Figure 14.11) is obtained by add- ing the operations (methods) to the class diagram of Figure 13.12. In the case of a Java implementation, two additional classes are needed. **Elevator**

**FIGURE 14.11** The detailed class diagram for the elevator problem case study. For clarity, only those methods that cause an object to change its state are shown.

|  |
| --- |
| **Elevator Application Class** |
|  |
|  |

|  |
| --- |
| **Button Class** |
| illuminated : Boolean |
| turnOffButton **(abstract)**  turnOnButton **(abstract)** |

|  |
| --- |
| **Elevator Utilities Class** |
|  |
|  |

turnOffButton turnOnButton

1..2

turnOffButton turnOnButton

m

controls

controls

1

2m — 2

communicates

with

1

communicates with

1

**Elevator Subcontroller Class**

1

1

controls

1

m communicates

with

n communicates

with

controls

1

1

1

moveDownOneFloor moveUpOneFloor

checkRequests updateRequests

requests: requestType

**Elevator Class**

**Scheduler Class**

startTimer

**r**

**ller Class**

**Floo Subcontro**

**Sensor Class**

**tton Class**

**Elevator Bu**

**on Class**

**Floor Butt**

|  |
| --- |
| **Elevator Doors Class** |
| doors open : Boolean |
| closeDoors openDoors |

**Application Class** corresponds to the C++ main function, and **Elevator Utilities Class** contains the Java routines that correspond to the C++ func- tions declared external to the C++ classes. (For clarity, methods of the form Send message to **C Class** . . . have been omitted from Figure 14.11; but see Problems 14.7–14.12.)

Consider the first iteration of the CRC card for the elevator subcontroller (Figure 13.14). The responsibilities fall into two groups. One responsibility—5. Start

timer—is assigned to the elevator controller on the basis of responsibility-driven design; that task is carried out by the elevator controller itself.

On the other hand, the remaining eleven responsibilities (events 1 through 4 and 6 through 12) have the form “Send a message to another class to tell it to do something.” This again implies that responsibility-driven design should be used in assigning the relevant method to classes. In addition, because of safety concerns, the principle of information hiding is equally applicable in all eleven cases.

For these two reasons, methods closeDoors and openDoors are assigned to **Elevator Doors Class**. That is, a client of **Elevator Doors Class** (in this case, an object of **Elevator Subcontroller Class**) sends a message to an object of **Elevator Doors Class** to close or open the doors of the elevator, and that request is then carried out by the relevant method. Every aspect of those two methods is encapsulated within **Elevator Doors Class**. In addition, information hiding results in a truly independent **Elevator Doors Class**, instances of which can undergo detailed design and implementation independently and be reused later in other products.

The same two design principles are applied to methods moveDownOneFloor and moveUpOneFloor, and they are assigned to **Elevator Class**. There is no need for an explicit instruction to cause an elevator to stop. If neither of its two methods is invoked, an elevator cannot move; there is no way to change the state of an elevator other than by invoking one of its two methods.

Finally, methods turnOffButton and turnOnButton are assigned to both **Ele- vator Button Class** and **Floor Button Class**. The reasoning here is the same as for the methods assigned to **Elevator Doors Class** and **Elevator Class**. First, the principle of responsibility-driven design requires that the buttons have full control over whether they are on or off. Second, the principle of information hiding requires the internal state of a button to be hidden. The methods that turn an elevator button on or off therefore must be local to **Elevator Button Class**, and similarly for **Floor Button Class**. To make use of polymorphism and dynamic binding, methods turnOffButton and turnOnButton are declared **abstract** (**virtual**) in the base class **Button Class** for the reasons stated in Section 7.8. At run time, the correct version of method turnOffButton or turnOnButton will then be invoked.

**Step 2. Perform the Detailed Design**

A detailed design now is developed for all the classes. Any suitable technique may be used, such as the stepwise refinement described in Chapter 5. The detailed design of method elevatorSubcontrollerEventLoop is shown in Figure 14.12. Here PDL (pseudocode) was used, but a tabular representation (such as that of Figure 14.6) can be equally effective.

Figure 14.12 is constructed from the statechart of Figure 13.13. For example, the events elevator button pushed and elevator button turned off is implemented by the two nested **if** statements at the beginning of Figure 14.12. The two operations

**FIGURE 14.12**

The detailed design of method elevator-

Subcontroller- EventLoop.

**void** elevatorSubcontrollerEventLoop (**void**)

{

**while** (TRUE)

{

**if** (*an* **elevatorButton** *has been pressed*)

**if** (**elevatorButton** *is off*)

{

**elevatorButton**::turnOnButton; **scheduler**::newRequestMade;

}

**else if** (**elevator** *is moving* up)

{

*wait for sensor message that elevator is arriving at floor;*

**scheduler**::checkRequests;

**if** (*there is no request to stop at floor* f)

**elevator**::moveUpOneFloor; **else**

{

*stop* **elevator** *by not sending a message to move;*

**if** (**elevatorButton** *is on*)

**elevatorButton**::turnOffButton; **elevatorDoors**::openDoors; startTimer;

}

}

**else if** (**elevator** *is moving* down) [*similar to* up *case*]

**else if** (**elevator** *is stopped and request is pending*)

{

*wait for timeout;* **elevatorDoors**::closeDoors; *determine direction of next request*; **elevator**::moveUp/DownOneFloor;

*wait for sensor message that elevator has left floor;*

**floorSubcontroller**::elevatorHasLeftFloor;

}

**else if** (**elevator** *is at rest* **and not** (*request is pending*))

{

*wait for timeout;*

**elevatorDoors**::closeDoors;

}

**else**

*there are no requests,* **elevator** *is stopped with* **elevatorDoors** *closed, so do nothing*;

}

}

of the state **Processing New Request** then follow. The **else-if** condition cor- responds to the next event leading from state **Elevator Subcontroller Event Loop**, elevator moving in direction d, floor f is next. The remainder of the detailed design is equally straightforward.

Now we consider the object-oriented design of the MSG Foundation case study.

###### Case Study

* 1. Object-Oriented Design: The MSG Foundation Case Study

As described in Section 14.6, object-oriented design consists of two steps.

**Step 1. Complete the Class Diagram**

The overall class diagram for the MSG Foundation case study is shown in Figure 14.13. The user-defined **Date Class** is drawn dashed to denote that it is needed for only

**FIGURE 14.13**

The overall class diagram for the MSG Foundation case study.

**Date Class**

**MSG Staff Member**

**User Interface Class**

**Manage an Asset Class**

**Estimate Funds for Week Class**

**Investment Class**



**Investments Report Class**

**Asset Class**

**Mortgage Class**

**Mortgages Report Class**

**MSG**

**Application Class**

**Estimated Funds Report Class**

a C++ implementation; Java has built-in classes for handling dates, including **java. text.Dateformat** and **java.util.Calendar**.

Next, the formats for the attributes of the classes are deduced from discussions with the client and users; examination of forms (Section 11.4.2) is also extremely useful in this regard. A portion of the result is shown in Figure 14.14.

The methods of the product are found in the various interaction diagrams. The task of the designer is to decide to which class each method should be assigned. For example, the convention in an object-oriented software product is that associated with each attribute of a class are **mutator** method setAttribute, used to assign a specific value to that attribute, and **accessor** method getAttribute, which returns the current value of that attribute.

For example, consider method setAssetNumber, used to assign a number to an asset (investment or mortgage). In the classical paradigm, we would need separate functions set\_investment\_number and set\_mortgage\_number. However, the object-oriented paradigm supports inheritance. Therefore, method setAssetNumber should be assigned to **Asset Class**. Then, as reflected in Figure 14.15, the method

**FIGURE 14.14** Part of the overall class diagram for the MSG Foundation case study with the attribute formats added.

|  |
| --- |
| **Asset Class** |
| assetNumber : 12 chars |
|  |

|  |
| --- |
| **MSG Application Class** |
| estimatedAnnualOperatingExpenses : 9 + 2 digits dateEstimatedAnnualOperatingExpensesUpdated : 10 chars availableFundsForWeek : 9 + 2 digits  expectedAnnualReturnOnInvestments : 9 + 2 digits dateExpectedAnnualReturnOnInvestmentsUpdated : 10 chars expectedGrantsForWeek : 9 + 2 digits  expectedMortgagePaymentsForWeek : 9 + 2 digits |
|  |

**Investment Class**

investmentName : 25 chars estimatedAnnualReturn : 9 digits dateEstimatedReturnUpdated : 10 chars

**Mortgage Class**

lastNameOfMortgagees : 21 chars originalPurchasePrice : 6 digits dateMortgageIssued : 10 chars weeklyPrincipalAndInterestPayment : 4 + 2 digits

combinedWeeklyIncome : 6 + 2 digits

mortgageBalance : 6 + 2 digits dateCombinedWeeklyIncomeUpdated : 10 chars annualRealEstateTax : 5 + 2 digits dateAnnualRealEstateTaxUpdated : 10 chars annualInsurancePremium : 5 + 2 digits dateAnnualInsurancePremiumUpdated : 10 chars

**FIGURE 14.15** Part of the class diagram for the MSG Foundation case study with methods

setAssetNumber and getAssetNumber assigned to **Asset Class**.

**Mortgage Class**

**Investment Class**

**MSG Application Class**

|  |
| --- |
| **Asset Class** |
|  |
| setAssetNumber ( ) getAssetNumber ( ) |

can be applied not only to instances of **Asset Class** but also, as a consequence of inheritance, to instances of every subclass of **Asset Class**, that is, to instances of **Investment Class** and **Mortgage Class**. Similarly, method getAssetNumber should also be allocated to the superclass **Asset Class**.

Assigning the other methods to the appropriate classes is equally straightforward.

The resulting design is shown in Appendix G.

**Step 2. Perform the Detailed Design**

Next, the detailed design is built by taking each method and determining what it does. Figure 14.16 shows the detailed design (in a PDL for Java) of a method com- puteEstimatedFunds of class **EstimateFundsForWeek** of the MSG Founda- tion case study. This method invokes method totalWeeklyNetPayments of class **Mortgage** shown in Figure 14.17.

The steps of object-oriented design are summarized in How to Perform Box 14.3.

* 1. **The Design Workflow**

The overall aim of the **design workflow** is to refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers. The input to the design workflow is therefore the analysis workflow artifacts (Chapter 13). During the design workflow, these artifacts are iterated and incremented until they are in a format that can be utilized by the programmers.

|  |
| --- |
| **How to Perform Object-Oriented Design Box 14.3** |
|  |
| * Complete the class diagram. * Perform the detailed design. |

**FIGURE 14.16**

The detailed design of method compute- Estimated- Funds of class **Estimate- FundsFor-**

**Week** of the MSG

Foundation case study.

**public static void** computeEstimatedFunds( )

*This method computes the estimated funds available for the week.*

{

**float** expectedWeeklyInvestmentReturn; (*expected weekly investment return*)

**float** expectedTotalWeeklyNetPayments = (**float**) 0.0;

(*expected total mortgage payments less total weekly grants*)

**float** estimatedFunds = (**float**) 0.0; (*total estimated funds for week*) *Create an instance of an investment record.*

Investment inv = **new** Investment ( );

*Create an instance of a mortgage record.*

Mortgage mort = **new** Mortgage ( );

*Invoke method* totalWeeklyReturnOnInvestment. expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );

*Invoke method* expectedTotalWeeklyNetPayments (*see Figure 14.17*) expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );

*Now compute the estimated funds for the week.*

estimatedFunds = (expectedWeeklyInvestmentReturn

– (MSGApplication.getAnnualOperatingExpenses ( ) / (**float**) 52.0)

+ expectedTotalWeeklyNetPayments);

*Store this value in the appropriate location.*

MSGApplication.setEstimatedFundsForWeek (estimatedFunds);

} // computeEstimatedFunds

One aspect of this iteration and incrementation is the identification of methods and their allocation to the appropriate classes. Another aspect is performing the detailed design. These two steps constitute the object-oriented design component of the design workflow.

In addition to performing the object-oriented design, many decisions have to be made as part of the design workflow. One such decision is the selection of the programming language in which the software product will be implemented. This process is described in detail in Chapter 15. Another decision is how much of existing software products to reuse in the new software product to be developed. Reuse is described in Chapter 8. Portability is another important design decision; this topic, too, is described in Chapter 8. Also, large software products are often implemented on a network of computers; yet another design decision is the allocation of each software component to the hardware component on which it is to run.

The major motivation behind the development of the Unified Process was to present a methodology that could be used to develop large-scale software products, typically, 500,000 lines of code or more. On the other hand, the implementations of the MSG Foundation case study in Appendices H and I are less than 5000 lines of C++ and Java, respectively. In other

**FIGURE 14.17**

The detailed design of method totalWeekly- NetPayments of class **Mortgage** of the MSG

Foundation case study.

**public float** totalWeeklyNetPayments ( )

*This method computes the net total weekly payments made by the mortgagees, that is, the expected total weekly mortgage amount less the expected total weekly grants.*

{

File mortgageFile = **new** File (“mortgage.dat”); (*file of mortgage records*)

**float** expectedTotalWeeklyMortgages = (**float**) 0.0; (*expected total weekly mortgage payments*)

**float** expectedTotalWeeklyGrants = (**float**) 0.0; (*expected total weekly grants*)

**float** interestPayment; (*interest payment*)

**float** escrowPayment; (*escrow payment*)

**float** capitalRepayment; (*capital repayment*)

**float** weeklyPayment; (*mortgage payment for week*)

**float** maximumPermittedMortgagePayment; (*maximum amount the couple may pay*) *Open the file of mortgages, name it* inFile*, and read each element in turn.*

{

read (inFile);

*Compute the interest payment, escrow payment, and capital repayment for this mortgage.* interestPayment = mortgageBalance \* INTEREST\_RATE / WEEKS\_IN\_YEAR ; escrowPayment = (annualPropertyTax + annualInsurancePremium) / WEEKS\_IN\_YEAR; capitalRepayment = weeklyPrincipalAndInterestPayment − interestPayment; mortgageBalance −= capitalRepayment;

*First assume that the couple can pay the mortgage in full, without a grant.*

weeklyPayment = weeklyPrincipalAndInterestPayment + escrowPayment;

*Add the weekly Principal and Interest payment to the running total of mortgage payments*

expectedTotalWeeklyMortgages += weeklyPrincipalAndInterestPayment;

*Now determine how much the couple can actually pay.*

maximumPermittedMortgagePayment = currentWeeklyIncome \* MAXIMUM\_PERC\_OF\_INCOME;

*If a grant is needed, add the grant amount to the running total of grants*

**if** (weeklyPayment > maximumPermittedMortgagePayment) expectedTotalWeeklyGrants += weeklyPayment − maximumPermittedMortgagePayment;

}

*Close the file of mortgages. Return the total expected net payments for the week.*

**return** (expectedTotalWeeklyMortgages − expectedTotalWeeklyGrants);

} // totalWeeklyNetPayments

words, the Unified Process is intended primarily for software products at least 100 times larger than the MSG Foundation case study presented in this book. Accordingly, many aspects of the Unified Process are inapplicable to this case study. For instance, an important part of the analysis workflow is to partition the software product into analysis packages. Each **package** consists of a set of related classes, usually of relevance to a small subset of the actors, that can be implemented as a single unit. For example, accounts payable, accounts receivable, and general ledger are typical analysis packages. The concept under- lying analysis packages is that it is much easier to develop smaller software products than larger software products. Accordingly, a large software product is easier to develop if it can be decomposed into relatively independent packages. Decomposing a software product into packages is an example of divide-and-conquer (Section 5.3).

This idea of decomposing a large workflow into relatively independent smaller work- flows is carried forward to the design workflow. Here, the objective is to break up the upcoming implementation workflow into manageable pieces, termed **subsystems**. Again, it does not make sense to break up the MSG Foundation case study into subsystems; the case study is just too small.

There are two reasons why larger workflows are broken into subsystems:

* + 1. As previously explained, it is easier to implement a number of smaller subsystems than one large system. That is, breaking up a software product into subsystems is another example of divide-and-conquer (Section 5.3).
    2. If the subsystems to be implemented are indeed relatively independent, then they can be implemented by programming teams working in parallel. This results in the software product as a whole being delivered sooner.

Recall from Section 8.5.4 that the *architecture* of a software product includes the vari- ous components and how they fit together. The allocation of components to subsystems is a major part of the architectural task. Deciding on the architecture of a software product is by no means easy and, in all but the smallest software products, is performed by a specialist, the software **architect**.

In addition to being a technical expert, an architect needs to know how to make **trade-offs**. A software product has to satisfy the functional requirements, that is, the use cases. It also needs to satisfy the nonfunctional requirements, including portability (Chapter 8), reliability (Section 6.4.2), robustness (Section 6.4.3), maintainability, and security. But it needs to do all these things within budget and time constraints. It is almost never possible to develop a software product that satisfies all its requirements, both functional and nonfunctional, and finish the project within the cost and time con- straints; compromises almost always have to be made. The client has to relax some of the requirements, increase the budget, or move the delivery deadline, or do more than one of these. The architect must assist the client’s decision making by clearly mapping out the trade-offs.

In some cases the trade-offs are obvious. For example, the architect may point out that a set of security requirements that conform to a new high-security standard are going to take a further 3 months and $350,000 to incorporate in the software product. If the product is an international banking network, the issue is moot—there is no way that the client could pos- sibly agree to compromise on security in any way. However, in other instances, the client needs to make critical determinations regarding trade-offs and has to rely on the technical

expertise of the architect to assist in coming to the right business decision. For example, the architect might point out that deferring a particular requirement until the software product has been delivered and is being maintained may save $150,000 now but will cost $300,000 to incorporate later (see Figure 1.6). The decision whether or not to defer a requirement can be made only by the client, but he or she needs the technical expertise of the architect to assist in coming to the correct decision.

The architecture of a software product is a vital factor in the delivered product’s success or a failure. And the critical decisions regarding the architecture have to be made while performing the design workflow. If the requirements workflow is badly performed, it is still possible to have a successful project, provided additional time and money are spent on the analysis workflow. Similarly, if the analysis workflow is inadequate, it is possible to recover by making an extra effort as part of the design workflow. But if the architecture is suboptimal, there is no way to recover; the architecture must immediately be redesigned. It is therefore essential that the development team include an architect with the necessary technical expertise and people skills.

(Dịch: Một khía cạnh của việc lặp lại và gia tăng này là việc xác định các phương thức và phân bổ chúng cho các lớp thích hợp. Một khía cạnh khác là thực hiện thiết kế chi tiết. Hai bước này cấu thành thành phần thiết kế hướng đối tượng của quy trình thiết kế.

Ngoài việc thực hiện thiết kế hướng đối tượng, nhiều quyết định phải được đưa ra như một phần của quy trình thiết kế. Một quyết định như vậy là việc lựa chọn ngôn ngữ lập trình mà sản phẩm phần mềm sẽ được triển khai. Quá trình này được mô tả chi tiết trong Chương 15. Một quyết định khác là sử dụng lại bao nhiêu sản phẩm phần mềm hiện có trong sản phẩm phần mềm mới sẽ được phát triển. Tái sử dụng được mô tả trong Chương 8. Tính di động là một quyết định thiết kế quan trọng khác; chủ đề này cũng được mô tả trong Chương 8. Ngoài ra, các sản phẩm phần mềm lớn thường được triển khai trên mạng máy tính; một quyết định thiết kế khác là phân bổ từng thành phần phần mềm cho thành phần phần cứng mà nó sẽ chạy trên đó.

Động lực chính đằng sau sự phát triển của Quy trình Hợp nhất là trình bày một phương pháp có thể được sử dụng để phát triển các sản phẩm phần mềm quy mô lớn, điển hình là 500.000 dòng mã trở lên. Mặt khác, việc triển khai nghiên cứu điển hình của Tổ chức MSG trong Phụ lục H và tôi lần lượt là ít hơn 5000 dòng của C++ và Java. Nói cách khác, Quy trình Thống nhất chủ yếu dành cho các sản phẩm phần mềm lớn hơn ít nhất 100 lần so với nghiên cứu tình huống của Tổ chức MSG được trình bày trong cuốn sách này. Theo đó, nhiều khía cạnh của Quy trình Thống nhất không thể áp dụng cho nghiên cứu điển hình này. Chẳng hạn, một phần quan trọng của quy trình phân tích là phân vùng sản phẩm phần mềm thành các gói phân tích. Mỗi gói bao gồm một tập hợp các lớp liên quan, thường liên quan đến một tập hợp con nhỏ của các tác nhân, có thể được triển khai như một đơn vị duy nhất. Ví dụ: tài khoản phải trả, tài khoản phải thu và sổ cái chung là các gói phân tích điển hình. Khái niệm nền tảng của các gói phân tích là việc phát triển các sản phẩm phần mềm nhỏ hơn sẽ dễ dàng hơn nhiều so với các sản phẩm phần mềm lớn hơn. Theo đó, một sản phẩm phần mềm lớn sẽ dễ phát triển hơn nếu nó có thể được phân tách thành các gói tương đối độc lập. Phân tách một sản phẩm phần mềm thành các gói là một ví dụ về chia để trị (Phần 5.3).

Ý tưởng phân tách một quy trình công việc lớn thành các quy trình công việc nhỏ hơn tương đối độc lập được chuyển sang quy trình thiết kế. Ở đây, mục tiêu là chia nhỏ quy trình triển khai sắp tới thành các phần có thể quản lý được, được gọi là các hệ thống con. Một lần nữa, sẽ không hợp lý nếu chia nghiên cứu tình huống của Tổ chức MSG thành các hệ thống con; nghiên cứu trường hợp chỉ là quá nhỏ.

Có hai lý do khiến quy trình công việc lớn hơn được chia thành các hệ thống con:

1. Như đã giải thích trước đây, việc triển khai một số hệ thống con nhỏ sẽ dễ dàng hơn so với một hệ thống lớn. Nghĩa là, chia nhỏ một sản phẩm phần mềm thành các hệ thống con là một ví dụ khác về chia để trị (Phần 5.3).

2. Nếu các hệ thống con được triển khai thực sự tương đối độc lập, thì chúng có thể được triển khai bởi các nhóm lập trình làm việc song song. Điều này dẫn đến toàn bộ sản phẩm phần mềm được chuyển giao sớm hơn.

Nhớ lại từ Phần 8.5.4 rằng kiến ​​trúc của một sản phẩm phần mềm bao gồm các thành phần khác nhau và cách chúng khớp với nhau. Việc phân bổ các thành phần cho các hệ thống con là một phần chính của nhiệm vụ kiến ​​trúc. Quyết định về kiến ​​trúc của một sản phẩm phần mềm không hề dễ dàng và, trong tất cả, trừ những sản phẩm phần mềm nhỏ nhất, được thực hiện bởi một chuyên gia, kiến ​​trúc sư phần mềm.

Ngoài việc là một chuyên gia kỹ thuật, một kiến ​​trúc sư cần biết cách đánh đổi. Một sản phẩm phần mềm phải đáp ứng các yêu cầu chức năng, nghĩa là các trường hợp sử dụng. Nó cũng cần đáp ứng các yêu cầu phi chức năng, bao gồm tính di động (Chương 8), độ tin cậy (Phần 6.4.2), độ bền (Phần 6.4.3), khả năng bảo trì và bảo mật. Nhưng nó cần phải làm tất cả những điều này trong giới hạn ngân sách và thời gian. Gần như không bao giờ có thể phát triển một sản phẩm phần mềm đáp ứng tất cả các yêu cầu của nó, cả chức năng và phi chức năng, và hoàn thành dự án trong các ràng buộc về chi phí và thời gian; thỏa hiệp hầu như luôn luôn phải được thực hiện. Khách hàng phải nới lỏng một số yêu cầu, tăng ngân sách hoặc dời thời hạn giao hàng hoặc thực hiện nhiều hơn một trong những yêu cầu này. Kiến trúc sư phải hỗ trợ quá trình ra quyết định của khách hàng bằng cách vạch ra rõ ràng những sự đánh đổi.

Trong một số trường hợp, sự đánh đổi là rõ ràng. Ví dụ, kiến ​​trúc sư có thể chỉ ra rằng một tập hợp các yêu cầu bảo mật phù hợp với tiêu chuẩn bảo mật cao mới sẽ mất thêm 3 tháng và 350.000 USD để đưa vào sản phẩm phần mềm. Nếu sản phẩm là một mạng lưới ngân hàng quốc tế, thì vấn đề sẽ được tranh luận – không có cách nào mà khách hàng có thể đồng ý thỏa hiệp về bảo mật theo bất kỳ cách nào. Tuy nhiên, trong các trường hợp khác, khách hàng cần đưa ra các quyết định quan trọng liên quan đến sự đánh đổi và phải dựa vào kỹ thuật. chuyên môn của kiến ​​trúc sư để hỗ trợ đưa ra quyết định kinh doanh đúng đắn. Ví dụ, kiến ​​trúc sư có thể chỉ ra rằng việc trì hoãn một yêu cầu cụ thể cho đến khi sản phẩm phần mềm được chuyển giao và đang được bảo trì có thể tiết kiệm được 150.000 đô la ngay bây giờ nhưng sẽ tốn 300.000 đô la để kết hợp sau này (xem Hình 1.6). Quyết định có trì hoãn yêu cầu hay không chỉ có thể được đưa ra bởi khách hàng, nhưng họ cần chuyên môn kỹ thuật của kiến ​​trúc sư để hỗ trợ đưa ra quyết định chính xác.

Kiến trúc của một sản phẩm phần mềm là một yếu tố sống còn trong sự thành công hay thất bại của sản phẩm được chuyển giao. Và các quyết định quan trọng liên quan đến kiến ​​trúc phải được đưa ra trong khi thực hiện quy trình thiết kế. Nếu quy trình làm việc yêu cầu được thực hiện kém, thì vẫn có thể có một dự án thành công, miễn là dành thêm thời gian và tiền bạc cho quy trình phân tích. Tương tự, nếu quy trình phân tích không đầy đủ, có thể khôi phục bằng cách nỗ lực thêm như một phần của quy trình thiết kế. Nhưng nếu kiến ​​trúc dưới mức tối ưu, thì không có cách nào để phục hồi; kiến trúc phải ngay lập tức được thiết kế lại. Do đó, điều cần thiết là nhóm phát triển phải bao gồm một kiến ​​trúc sư có chuyên môn kỹ thuật và kỹ năng con người cần thiết.

* 1. The Test Workflow: Design 14.10 Quy trình kiểm thử: Thiết kế

The goal of testing the design is to verify that the specifications have been accurately and completely incorporated into the design as well as to ensure the correctness of the design itself. For example, the design must have no logic faults, and all interfaces must be cor- rectly defined. It is important that any faults in the design be detected before coding com- mences; otherwise, the cost of fixing the faults will be considerably higher, as reflected in Figure 1.6. Design faults can be detected by means of design inspections as well as design walkthroughs. Design inspections are discussed in the remainder of this section, but the remarks apply equally to design walkthroughs.

When the product is transaction oriented (Section 14.4), the design inspection should reflect this [Beizer, 1990]. Inspections that include all possible transaction types should be scheduled. The reviewer should relate each transaction in the design to the specifications, showing how the transaction arises from the specification document. For example, if the application is an automated teller machine, a transaction corresponds to each operation the customer can perform, such as deposit to or withdraw from a credit card account. In other instances, the correspondence between specifications and transactions is not necessarily one-to-one. In a traffic-light control system, for example, if an automobile driving over a sensor pad results in the system deciding to change a particular light from red to green in 15 seconds, then further impulses from that sensor pad may be ignored. Conversely, to speed traffic flow, a single impulse may cause a whole series of lights to be changed from red to green.

Restricting reviews to **transaction-driven inspections** does not detect cases where the designers have overlooked instances of transactions required by the specifications. To take an extreme example, the specifications for the traffic-light controller may stipulate that between 11:00 P.M. and 6:00 A.M. all lights are to flash yellow in one direction and red in the other direction. If the designers overlooked this stipulation, then clock-generated transactions at 11:00 P.M. and 6:00 A.M. would not be included in the design; and if these transactions were overlooked, they could not be tested in a design inspection based on

###### ase Study

*C*

transactions. Therefore, it is not adequate to schedule design inspections that are just trans- action driven; specification-driven inspections also are essential to ensure that no statement in the specification document has been either overlooked or misinterpreted.

The Test Workflow: The MSG Foundation Case Study



Now that the design is apparently complete, all aspects of the design of the MSG Foundation case study must be checked by means of a design inspection (Section 6.2.3). In particular, each design artifact must be examined. Even if no faults are found, it is possible that the design will change again, perhaps radically, when the MSG Foundation case study is implemented.

Quy trình làm việc thử nghiệm: Nghiên cứu điển hình về nền tảng MSG

Giờ đây, khi thiết kế rõ ràng đã hoàn tất, tất cả các khía cạnh của thiết kế trong nghiên cứu tình huống của Tổ chức MSG phải được kiểm tra bằng phương pháp kiểm tra thiết kế (Phần 6.2.3). Đặc biệt, mỗi hiện vật thiết kế phải được kiểm tra. Ngay cả khi không tìm thấy lỗi nào, có thể thiết kế sẽ thay đổi một lần nữa, có lẽ là hoàn toàn, khi nghiên cứu điển hình của Tổ chức MSG được triển khai

* 1. **Formal Techniques for Detailed Design :** **14.12 Kỹ thuật chính thức cho thiết kế chi tiết**

One technique for detailed design has already been presented. In Section 5.1, a description of stepwise refinement was given. It then was applied to detailed design using flowcharts. In addition to stepwise refinement, formal techniques can be used to advantage in detailed design. Chapter 6 suggests that implementing a complete product and then proving it cor- rect could be counterproductive. However, developing the proof and the detailed design in parallel and carefully testing the code as well is quite a different matter. Formal techniques applied to detailed design can greatly assist in three ways:

* + 1. The state of the art in proving correctness is such that, although it generally cannot be applied to a product as a whole, it can be applied to module-sized pieces of a product.
    2. Developing a proof together with the detailed design should lead to a design with fewer faults than if correctness proofs were not used.
    3. If the same programmer is responsible for both the detailed design and the implementa- tion, then that programmer will feel confident that the detailed design is correct. This positive attitude toward the design should lead to fewer faults in the code.
  1. Real-Time Design Techniques

As explained in Section 6.4.4, **real-time software** is characterized by hard time con- straints, that is, time constraints of such a nature that, if a constraint is not met, informa- tion is lost. In particular, each input must be processed before the next input arrives. An example of such a system is a computer-controlled nuclear reactor. Inputs such as the temperature of the core and the level of the water in the reactor chamber are continually being sent to the computer that reads the value of each input and performs the necessary

processing before the next input arrives. Another example is a computer-controlled inten- sive care unit. There are two types of patient data: routine information such as heart rate, temperature, and blood pressure of each patient, and emergency information, when the system deduces that the condition of a patient has become critical. When such emergencies occur, the software must process both the routine inputs and the emergency-related inputs from one or more patients.

A characteristic of many real-time systems is that they are implemented on distributed hardware. For example, software controlling a fighter aircraft may be implemented on five computers: one to handle navigation, another the weapons system, a third for electronic coun- termeasures, a fourth to control the flight hardware such as wing flaps and engines, and the fifth to propose tactics in combat. Because hardware is not totally reliable, there may be addi- tional backup computers that automatically replace a malfunctioning unit. Not only does the design of such a system have major communications implications, but timing issues, over and above those of the type just described, arise as a consequence of the distributed nature of the system. For example, under combat conditions, the tactical computer might suggest that the pilot should climb, whereas the weapons computer recommends that the pilot go into a dive so that a particular weapon may be launched under optimal conditions. However, the human pilot decides to move the stick to the right, thereby sending a signal to the flight hardware computer to make the necessary adjustments so that the plane banks in the indicated direc- tion. All this information must be managed carefully in such a way that the actual motion of the plane takes precedence in every way over suggested maneuvers. Furthermore, the actual motion must be relayed to the tactical and weapons computers so that new suggestions can be formulated in the light of actual, rather than suggested, conditions.

A further difficulty with real-time systems is the problem of synchronization. Suppose that a real-time system is to be implemented on distributed hardware. Situations such as deadlock (or deadly embrace) can arise when two operations each have exclusive use of a data item and each requests exclusive use of the other’s data item in addition. Of course, deadlock does not occur only in real-time systems, implemented on distributed hardware. But it is particu- larly troublesome in real-time systems where there is no control over the order or timing of the inputs, and the situation can be complicated by the distributed nature of the hardware. In addition to deadlock, other synchronization problems are possible, including race conditions; for details, the reader may refer to [Silberschatz, Galvin, and Gagne, 2002] or other operating systems textbooks.

From these examples it is clear that the major difficulty with regard to the design of real- time systems is ensuring that the timing constraints are met by the design. That is, the design technique should provide a mechanism for checking that, when implemented, the design is able to read and process incoming data at the required rate. Furthermore, it should be pos- sible to show that synchronization issues in the design also have been addressed correctly.

Since the beginning of the computer age, advances in hardware technology have out- stripped, in almost every respect, advances in software technology. Therefore, although the hardware exists to handle every aspect of the real-time systems described previously, soft- ware design technology has lagged behind considerably. In some areas of real-time software engineering, major progress has been made. For instance, many of the analysis techniques of Chapters 12 and 13 can be used to specify real-time systems. Unfortunately, software design has not yet reached the same level of sophistication. Great strides indeed are being made, but the state of the art is not yet comparable to what has been achieved with regard to analysis

techniques. Because almost any design technique for real-time systems is preferable to no technique at all, a number of real-time design techniques are used in practice. But, there still is a long way to go before it will be possible to design real-time systems such as those described previously and be certain that, before the system has been implemented, every real-time constraint will be met and synchronization problems cannot arise.

Older real-time design techniques are extensions of non-real-time techniques to the real-time domain. For example, structured development for real-time systems (SDRTS) [Ward and Mellor, 1985] essentially is an extension of structured systems analysis (Section 12.3), data flow analysis (Section 14.3), and transaction analysis (Section 14.4) to real-time software. The development technique includes a component for real-time design. Newer techniques are described in [Liu, 2000] and [Gomaa, 2000].

As stated previously, it is unfortunate that the state of the art of real-time design is not as advanced as one would wish. Nevertheless, efforts are under way to improve the situation.

* 1. CASE Tools for Design

As stated in Section 14.10, a critical aspect of design is testing that the design artifacts accurately incorporate all aspects of the analysis. What is therefore needed is a CASE tool that can be used both for the analysis artifacts and the design artifacts, a so-called front-end or upperCASE tool (as opposed to a back-end or lowerCASE tool, which assists with the implementation artifacts).

A number of upperCASE tools are on the market. Some of the more popular ones include Analyst/Designer, Software through Pictures, and System Architect. UpperCASE tools generally are built around a data dictionary. The CASE tool can check that every field of every record in the dictionary is mentioned somewhere in the design or that every item in the design is reflected in the data flow diagram. In addition, many upperCASE tools incorporate a consistency checker that uses the data dictionary to determine that every item in the design has been declared in the specifications and conversely that every item in the specifications appears in the design.

Furthermore, many upperCASE tools incorporate screen and report generators. That is, the client can specify what items are to appear in a report or on an input screen and where and how each item is to appear. Because full details regarding every item are in the data dictionary, the CASE tool can easily generate the code for printing the report or displaying the input screen according to the client’s wishes. Some upperCASE products also incorpo- rate management tools for estimating and planning.

With regard to object-oriented design, Together, IBM Rational Rose, and Software through Pictures provide support for this workflow within the context of the complete object-oriented life cycle. Open-source CASE tools of this type include ArgoUML.

* 1. Metrics for Design

A variety of metrics can be used to describe aspects of the design. For example, the number of code artifacts (modules or classes) is a crude measure of the size of the target product. Cohesion and coupling are measures of the quality of the design, as are fault statistics. As with all other types of inspection, it is vital to keep a record of the number and type

of design faults detected during a design inspection. This information is used during code inspections of the product and in design inspections of subsequent products.

The **cyclomatic complexity** *M* of a detailed design is the number of binary decisions (predicates) plus 1 [McCabe, 1976] or, equivalently, the number of branches in the code arti- fact. It has been suggested that cyclomatic complexity is a metric of design quality; the lower the value of *M*, the better. A strength of this metric is that it is easy to compute. However, it has an inherent problem. Cyclomatic complexity is purely a measure of the control complex- ity; the data complexity is ignored. That is, *M* does not measure the complexity of a code artifact that is data driven, such as by the values in a table. For example, suppose a designer is unaware of the C++ library function toascii and designs a code artifact from scratch that reads a character input by the user and returns the corresponding ASCII code (an integer between 0 and 127). One way of designing this is by means of a 128-way branch implemented by means of a **switch** statement. A second way is to have an array containing the 128 char- acters in ASCII code order and utilize a loop to compare the character input by the user with each element of the array of characters; the loop is exited when a match is obtained. The current value of the loop variable then is the corresponding ASCII code. The two designs are equivalent in functionality but have cyclomatic complexities of 128 and 1, respectively.

When the classical paradigm is used, a related class of metrics for the design phase is based on representing the architectural design as a directed graph with the modules repre- sented by nodes and the flows between modules (procedure and function calls) represented by arcs. The **fan-in** of a module can be defined as the number of flows into the module plus the number of global data structures accessed by the module. The **fan-out** similarly is the number of flows out of the module plus the number of global data structures updated by the module. A measure of complexity of the module then is given by *length × (fan-in × fan-out)*2 [Henry and Kafura, 1981], where **length** is a measure of the size of the module (Section 9.2.1). Because the definitions of *fan-in* and *fan-out* incorporate global data, this metric has a data-dependent component. Nevertheless, experiments have shown that this metric is no better a measure of complexity than simpler metrics, such as cyclomatic com- plexity [Kitchenham, Pickard, and Linkman, 1990; Shepperd, 1990].

The issue of design metrics is complicated even more when the object-oriented paradigm is used. For example, the cyclomatic complexity of a class usually is low, because many classes typically include a large number of small, straightforward methods. Furthermore, as previously pointed out, cyclomatic complexity ignores data complexity. Because data and operations are equal partners within the object-oriented paradigm, cyclomatic complexity overlooks a major component that could contribute to the complexity of an object. There- fore, metrics for classes that incorporate cyclomatic complexity generally are of little use.

A number of object-oriented design metrics have been put forward, for example, in [Chidamber and Kemerer, 1994]. These and other metrics have been questioned on both theoretical and experimental grounds [Binkley and Schach, 1996; 1997; 1998].

* 1. Challenges of the Design Workflow

As pointed out in Sections 12.16 and 13.22, it is important not to do too much in the analysis workflow; that is, the analysis team must not prematurely start parts of the design workflow. In the design workflow, the design team can go wrong in two ways: by doing too much and by doing too little.

Consider the PDL (pseudocode) detailed design of Figure 14.7. The temptation is strong for a designer who enjoys programming to write the detailed design in C++ or Java, rather than PDL. That is, instead of sketching the detailed design in pseudocode, the designer may all but code the class. This takes longer to write than just outlining the class and longer to fix if a fault is detected in the design (see Figure 1.6). Like the analysis team, the members of the design team must firmly resist the urge to do more than what is required of them.

At the same time, the design team must be careful not to do too little. Consider the tabu- lar detailed design of Figure 14.6. If the design team is in a hurry, it may decide to shrink the detailed design to just the narrative box. The team may even decide that the program- mers should do the detailed design by themselves. Either of these decisions would be a mistake. A primary reason for the detailed design is to ensure that all interfaces are correct. The narrative box by itself is inadequate for this purpose; no detailed design at all clearly is even less helpful. Therefore, one challenge of the design workflow is for the designers to do just the correct amount of work.

In addition, there is a much more significant challenge. In “No Silver Bullet” (see Just in Case You Wanted to Know Box 3.4), Brooks [1986] decries the lack of what he terms *great designers*, that is, designers who are significantly more outstanding than the other members of the design team. In Brooks’s opinion, the success of a software project depends critically on whether the design team is led by a great designer. Good design can be taught; great design is produced only by great designers, and they are “very rare.”

The challenge, then, is to grow great designers. They should be identified as early as possible (the best designers are not necessarily the most experienced), assigned a mentor, provided a formal education as well as apprenticeships to great designers, and allowed to interact with other designers. A specific career path should be available for these design- ers, and the rewards they receive should be commensurate with the contribution that only a great designer can make to a software development project.

**Chapter Review**

The design workflow is introduced in Section 14.1. There are three basic approaches to design: operation- oriented design (Section 14.2), data-oriented design (Section 14.5), and object-oriented design (Sec- tion 14.6). Two instances of operation-oriented design are described, data flow analysis (Section 14.3) and transaction analysis (Section 14.4). Object-oriented design is applied to the elevator problem case study in Section 14.7 and to the MSG Foundation case study in Section 14.8. The design workflow is presented in Section 14.9. The design aspects of the test workflow are described in Section 14.10 and applied to the MSG Foundation case study in Section 14.11. Formal techniques for detailed design are discussed in Section 14.12. Real-time system design is described in Section 14.13. CASE tools and metrics for the design workflow are presented in Sections 14.14 and 14.15, respectively. The chapter concludes with a discussion of the challenges of the design workflow (Section 14.16).

An overview of the MSG Foundation case study for Chapter 14 appears in Figure 14.18, and for the elevator problem in Figure 14.19.

**FIGURE 14.18**

|  |
| --- |
| Object-oriented design Section 14.8  Overall class diagram Figure 14.13  Part of overall class diagram Figure 14.14  with attribute formats added  Detailed design Appendix G |

Overview of the MSG Foundation case study for Chapter 14.

**FIGURE 14.19** Overview of the elevator problem case study for Chapter 14.

Object-oriented design Detailed class diagram

Section 14.7

Figure 14.11

**For Further Reading**

Data flow analysis and transaction analysis are described in books such as [Gane and Sarsen, 1979] and [Yourdon and Constantine, 1979].

The March–April 2005 issue of *IEEE Software* contains a number of papers on design. Designing for recovery, that is, designing software to detect, react, and recover from exceptional conditions, is described in [Wirfs-Brock, 2006].

Briand, Bunse, and Daly [2001] discuss the maintainability of object-oriented designs. A com- parison of both object-oriented and classical design techniques appears in [Fichman and Kemerer, 1992]. The redesign of an air traffic control system is described in [Jackson and Chapin, 2000]. Design techniques for high-performance, reliable systems are given in [Stolper, 1999]. A probabilis- tic approach to estimating the change proneness of an object-oriented design appears in [Tsantalis, Chatzigeorgiou, and Stephanides, 2005]. A discussion as to whether object-oriented design is intui- tive appears in [Hadar and Leron, 2008].

Formal design techniques are described in [Hoare, 1987]. The vital role played by the architect is described in [McBride, 2007]. Analogously to pair programming, pair design and its effectiveness are described in [Lui, Chan, and Nosek, 2008].

With regard to reviews during the design process, the original paper on design inspections is [Fagan, 1976]; detailed information can be obtained from that paper. Later advances in review tech- niques are described in [Fagan, 1986]. Architecture reviews are discussed in [Maranzano et al., 2005].

With regard to real-time design, specific techniques are to be found in [Liu, 2000] and [Gomaa, 2000]. A comparison of four real-time design techniques is found in [Kelly and Sherif, 1992]. A documentation-driven approach to the design of complex real-time systems is described in [Luqi, Zhang, Berzins, and Qiao, 2004]. The design of concurrent systems is described in [Magee and Kramer, 1999].

Metrics for design are described in [Henry and Kafura, 1981] and [Zage and Zage, 1993]. Metrics for object-oriented design are discussed in [Chidamber and Kemerer, 1994] and in [Binkley and Schach, 1996]. A model for object-oriented quality is presented in [Bansiya and Davis, 2002].

The proceedings of the International Workshops on Software Specification and Design are a com- prehensive source for information on design techniques.

**Key Terms** abstract data type design *476*

accessor *482*

architect *486*

architectural design *466*

class diagram *476*

cyclomatic complexity *491* data flow analysis (DFA) *467* data-oriented design *465*

design workflow *483*

detailed design *466*

fan-in *491*

fan-out *491*

general design *466*

high-level design *466*

length *491*

logical design *466*

low-level design *466*

modular design *466*

mutator *482*

object-oriented design (OOD)

*476*

operation-oriented design *465*

package *486*

physical design *466*

**494** Part B *The Workflows of the Software Life Cycle*

point of highest abstraction of input *467*

point of highest abstraction of output *467*

program description language (PDL) *471*

pseudocode *471*

real-time software *488*

responsibility-driven design

*477*

subsystem *486*

trade-off *486*

transaction *473*

transaction analysis *475*

transaction-driven inspections

*487*

**Problems** 14.1 Starting with your DFD for Problem 12.9, use data flow analysis to design a product for determining whether a bank statement is correct.

* 1. Use transaction analysis to design the software to control an ATM (Problem 8.9). At this stage omit error-handling capabilities.
  2. Now take your design for Problem 14.2 and add modules to perform error handling. Carefully examine the resulting design and determine the cohesion and coupling of the modules. Be on the lookout for situations such as that depicted in Figure 14.10.
  3. Two different techniques for depicting a detailed design are presented in Section 14.3.1 (Figures 14.6 and 14.7). Compare and contrast the two techniques.
  4. Starting with your data flow diagram for the automated library circulation system (Problem 12.11), design the circulation system using data flow analysis.
  5. Repeat Problem 14.5 using transaction analysis. Which of the two techniques did you find to be more appropriate?
  6. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Elevator Subcontroller Class**.
  7. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Floor Subcontroller Class**.
  8. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Sensor Class**.
  9. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Floor Button Class**.
  10. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Elevator Button Class**.
  11. Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Scheduler Class**.
  12. (Analysis and Design Project) Starting with your object-oriented analysis for the automated library circulation system (Problem 13.19), design the library system using object-oriented design.
  13. (Analysis and Design Project) Starting with your object-oriented analysis for the product for determining whether a bank statement is correct (Problem 13.20), design the software using object-oriented design.
  14. (Analysis and Design Project) Starting with your object-oriented analysis for the ATM soft- ware (Problem 13.21), design the ATM software using object-oriented design.

Chapter 14 *Design* **495**

* 1. (Term Project) Starting with your specifications of Problem 12.20 or 13.22, design the Chocoholics Anonymous product (Appendix A). Use the design technique specified by your instructor.
  2. (Case Study) Redesign the MSG Foundation product using data flow analysis.
  3. (Case Study) Redesign the MSG Foundation product using transaction analysis.
  4. (Case Study) The detailed design of Figures 14.16 and 14.17 is represented in PDL form. Represent the design using a tabular format. Which representation is superior? Give reasons for your answer.
  5. (Readings in Software Engineering) Your instructor will distribute copies of [Hadar and Leron, 2008]. To what extent do you think that object-oriented design is intuitive?

**References** [Bansiya and Davis, 2002] J. BANSIYA AND C. G. DAVIS, “A Hierarchical Model for Object-Oriented Design Quality Assessment,” *IEEE Transactions on Software Engineering* **28** (January 2002), pp. 4–17.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques,* 2nd ed., Van Nostrand Reinhold, New York, 1990.

[Binkley and Schach, 1996] A. B. BINKLEY AND S. R. SCHACH, “A Comparison of Sixteen Quality Metrics for Object-Oriented Design,” *Information Processing Letters* **57** (No. 6, June 1996), pp. 271–75.

[Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, “Toward a Unified Approach to Object- Oriented Coupling,” *Proceedings of the 35th Annual ACM Southeast Conference*, Murfreesboro, TN, April 2-4, 1997, IEEE, pp. 91–97.

[Binkley and Schach, 1998] A. B. BINKLEY AND S. R. SCHACH, “Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures,” *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1988, IEEE, pp. 542–55.

[Briand, Bunse, and Daly, 2001] L. C. BRIAND, C. BUNSE, AND J. W. DALY, “A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs,” *IEEE Transactions on Software Engineering* **27** (June 2001), pp. 513–30.

[Brooks, 1986] F. P. BROOKS, JR., “No Silver Bullet,” in: *Information Processing ’86*, H.-J. Kugler (Editor), Elsevier North-Holland, New York, 1986; reprinted in: *IEEE Computer* **20** (April 1987), pp. 10–19.

[Chidamber and Kemerer, 1994] S. R. CHIDAMBER AND C. F. KEMERER, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–93.

[Fagan, 1976] M. E. FAGAN, “Design and Code Inspections to Reduce Errors in Program Develop- ment,” *IBM Systems Journal* **15** (No. 3, 1976), pp. 182–211.

[Fagan, 1986] M. E. FAGAN, “Advances in Software Inspections,” *IEEE Transactions on Software Engineering* **SE-12** (July 1986), pp. 744–51.

[Fichman and Kemerer, 1992] R. G. FICHMAN AND C. F. KEMERER, “Object-Oriented and Con- ventional Analysis and Design Methodologies: Comparison and Critique,” *IEEE Computer* **25** (October 1992), pp. 22–39.

[Flanagan, 2005] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O’Reilly and Associates, Sebastopol, CA, 2005.

[Gane and Sarsen, 1979] C. GANE AND T. SARSEN, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[Goldberg and Robson, 1989] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language,* Addison- Wesley, Reading, MA, 1989.

**496** Part B *The Workflows of the Software Life Cycle*

[Gomaa, 2000] H. GOMAA, *Designing Concurrent, Distributed, and Real-time Applications with UML*, Addison-Wesley, Reading, MA, 2000.

[Hadar and Leron, 2008] “How Intuitive Is Object-Oriented Design?” *Communications of the ACM*

**51** (May 2008), pp. 41–46.

[Henry and Kafura, 1981] S. M. HENRY AND D. KAFURA, “Software Structure Metrics Based on Infor- mation Flow,” *IEEE Transactions on Software Engineering* **SE-7** (September 1981), pp. 510–18. [Hoare, 1987] C. A. R. HOARE, “An Overview of Some Formal Methods for Program Design,” *IEEE*

*Computer* **20** (September 1987), pp. 85–91.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1995.

[Jackson, 1975] M. A. JACKSON, *Principles of Program Design*, Academic Press, New York, 1975. [Jackson and Chapin, 2000] D. JACKSON AND J. CHAPIN, “Redesigning Air Traffic Control: An Exer-

cise in Software Design,” *IEEE Software* **17** (May–June 2000), pp. 63–70.

[Kelly and Sherif, 1992] J. C. KELLY AND J. S. SHERIF, “A Comparison of Four Design Methods for Real-Time Software Development,” *Information and Software Technology* **34** (February 1992), pp. 74–82.

[Kitchenham, Pickard, and Linkman, 1990] B. A. KITCHENHAM, L. M. PICKARD, AND S. J. LINK- MAN, “An Evaluation of Some Design Metrics,” *Software Engineering Journal* **5** (January 1990), pp. 50–58.

[Liu, 2000] J. W. S. LIU, *Real Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[Lui, Chan, and Nosek, 2008] K. M. LUI, K. C. C. CHAN, AND J. T. NOSEK, “The Effect of Pairs in Program Design Tasks,” *IEEE Transactions on Software Engineering* **34** (March–April 2008), pp. 197–211.

[Luqi, Zhang, Berzins, and Qiao, 2004] LUQI, L. ZHANG, V. BERZINS, AND Y. QIAO, “Documentation Driven Development for Complex Real-Time Systems,” *IEEE Transactions on Software Engi- neering* **30** (December 2004), pp. 936–52.

[Magee and Kramer, 1999] J. MAGEE AND J. KRAMER, *Concurrency: State Models & Java Programs*, John Wiley and Sons, New York, 1999.

[Maranzano et al., 2005] J. F. MARANZANO, S. A. ROZSYPAL, G. H. ZIMMERMAN, G. W. WARNKEN, P.

E. WIRTH, AND D. M. WEISS, “Architecture Reviews: Practice and Experience,” *IEEE Software* **22**

(March–April 2005), pp. 34–43.

[McCabe, 1976] T. J. MCCABE, “A Complexity Measure,” *IEEE Transactions on Software Engineer- ing* **SE-2** (December 1976), pp. 308–20.

[McBride, 2007] M. R. MCBRIDE, “The Software Architect,” *Communications of the ACM* **50** (May 2007), pp. 75–81.

[Orr, 1981] K. ORR, *Structured Requirements Definition*, Ken Orr and Associates, Topeka, KS, 1981. [Shepperd, 1990] M. SHEPPERD, “Design Metrics: An Empirical Analysis,” *Software Engineering*

*Journal* **5** (January 1990), pp. 3–10.

[Silberschatz, Galvin, and Gagne, 2002] A. SILBERSCHATZ, P. B. GALVIN, AND G. GAGNE, *Operating System Concepts,* 6th ed., Addison-Wesley, Reading, MA, 2002.

[Stolper, 1999] S. A. STOLPER, “Streamlined Design Approach Lands Mars Pathfinder,” *IEEE Soft- ware* **16** (September–October 1999), pp. 52–62.

[Stroustrup, 2003] B. STROUSTRUP, *The C++ Standard: Incorporating Technical Corrigendum No. 1*, 2nd ed., John Wiley and Sons, New York, 2003.

Chapter 14 *Design* **497**

[Tsantalis, Chatzigeorgiou, and Stephanides, 2005] N. TSANTALIS, A. CHATZIGEORGIOU, AND G. STEPHANIDES, “Predicting the Probability of Change in Object-Oriented Systems,” *IEEE Transac- tions on Software Engineering* **31** (July 2005), pp. 601–14.

[Ward and Mellor, 1985] P. T. WARD AND S. MELLOR, *Structured Development for Real-Time Systems,*

Vols. 1, 2, and 3, Yourdon Press, New York, 1985.

[Warnier, 1976] J. D. WARNIER, *Logical Construction of Programs*, Van Nostrand Reinhold, New York, 1976.

[Wirfs-Brock, 2006] R. WIRFS-BROCK, “Designing for Recovery,” *IEEE Software* **23** (July–August 2006), pp. 11–13.

[Yourdon and Constantine, 1979] E. YOURDON AND L. L. CONSTANTINE, *Structured Design: Funda- mentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[Zage and Zage, 1993] W. M. ZAGE AND D. M. ZAGE, “Evaluating Design Metrics on Large-Scale Software,” *IEEE Software* **10** (July 1993), pp. 75–81.