Chapter 13

## Object-Oriented Analysis

**Learning Objectives**

After studying this chapter, you should be able to

* Perform the analysis workflow.
* Extract the boundary, control, and entity classes.
* Perform functional modeling.
* Perform class modeling.
* Perform dynamic modeling.
* Perform use-case realization.

In Chapter 12, we examined various classical analysis techniques. This chapter is the object-oriented counterpart of Chapter 12.

**Object-oriented analysis (OOA)** is a semiformal analysis technique for the object- oriented paradigm. In Chapter 12, we pointed out that a number of different techniques are used for structured systems analysis, all essentially equivalent. Similarly, well over 60 different techniques have been put forward for OOA. Again, all the techniques are largely equivalent. The “For Further Reading” section of this chapter includes references to a wide variety of techniques, as well as to published comparisons of different techniques.

However, as explained in Section 3.1, today the Unified Process [Jacobson, Booch, and Rumbaugh, 1999] is almost always the methodology of choice for object-oriented software production. For this reason, the first and last parts of this chapter are devoted to the analysis workflow of the Unified Process.

Object-oriented analysis is a key component of the object-oriented paradigm. When this workflow is performed, the classes are extracted. The use cases and the classes are the basis

**404**



Chapter 13 *Object-Oriented Analysis* **405**

**Just in Case You Wanted to Know Box 13.1**

Most of the major advances in the object-oriented paradigm were made between 1990 and 1995. Because it usually takes some 15 years for new technology to become accepted, wide- spread adoption of the object-oriented paradigm should have started no sooner than 2005. However, the **millennium bug** or **Y2K problem** changed the expected timetable.

In the 1960s, when computers first started to be used for business on a widespread basis, hardware was far more expensive than it is today. As a result, the vast majority of software products of that vintage represented a date using only the last two digits for a year; the leading 19 was understood. The problem with this scheme is that the year 00 is then inter- preted as 1900, not 2000.

When hardware became cheaper in the 1970s and 1980s, few managers saw any point in spending large sums of money rewriting existing software products with four-digit dates. After all, by the time the year 2000 arrived, it would be someone else’s problem. As a result, **legacy systems** remained year-2000 noncompliant. However, as the deadline of January 1, 2000, neared, software organizations were forced to work against the clock to fix their soft- ware products; there was no way to postpone the arrival of Y2K.

Problems facing the maintenance programmers included a lack of documentation for many legacy software products, as well as software products implemented in program- ming languages that were now obsolete. When modifying an existing software product was impossible, the only alternative was to start again from scratch. Some companies decided to use COTS technology (Section 1.11). Others decided that new custom software products were needed. For obvious reasons, managers wanted these software products to be devel- oped using modern technology that had already been shown to be cost effective, and that meant using the object-oriented paradigm. The Y2K problem was therefore a significant catalyst for the widespread acceptance of the object-oriented paradigm.

of the object-oriented software product to be developed. (For more insight into the object- oriented paradigm, see Just in Case You Wanted to Know Box 13.1.)

* 1. **The Analysis Workflow**

The **analysis workflow** of the Unified Process [Jacobson, Booch, and Rumbaugh, 1999] has two overall aims. From the viewpoint of the requirements workflow (the preceding workflow), the aim of the analysis workflow is to obtain a deeper understanding of the requirements. Con- versely, from the viewpoint of the design and implementation workflows (the workflows that follow the analysis workflow), the aim of the analysis workflow is to describe those require- ments in such a way that the resulting design and implementation are easy to maintain.

The Unified Process is use-case driven. During the analysis workflow, the use cases are described in terms of the classes of the software product. The Unified Process has three types of classes: entity classes, boundary classes, and control classes. An **entity class** models information that is long lived. In the case of a banking software product, **Account Class** is an entity class because information on accounts has to stay in the software prod- uct. For the MSG Foundation software product, **Investment Class** is an entity class; again, information on investments has to be long lived.

A **boundary class** models the interaction between the software product and its actors. Boundary classes are generally associated with input and output. For example, in the MSG

**FIGURE 13.1** UML stereotypes (extensions of UML) for representing an entity class, a boundary class, and a control class.

**Entity Class Boundary Class Control Class**

Foundation software product, reports have to be printed listing the investments of the Foundation, as well as all the mortgages currently held. This means that boundary classes **Investments Report Class** and **Mortgages Report Class** are needed.

A **control class** models complex computations and algorithms. In the case of the MSG Foundation software product, the algorithm for estimating the funds available for the week is a control class, namely, **Estimate Funds for Week Class**.

The UML notation for these three types of classes is shown in Figure 13.1. These are **stereotypes**, that is, extensions of UML. A strength of UML is that it allows additional constructs to be defined that are not part of UML but may be needed to model a specific system accurately.

As stated at the beginning of this section, during the analysis workflow, the use cases are described in terms of the classes of the software product. The Unified Pro- cess itself does not describe how classes are to be extracted because users of the Unified Process are expected to have a background in object-oriented analysis and design. Accordingly, this discussion of the Unified Process is temporarily suspended so that an explanation can be given of how classes are extracted; we return to the Uni- fied Process in Section 13.15.

Entity classes, that is, classes that model long-lived information, are considered first.

(Dịch: Quy trình phân tích của Quy trình hợp nhất [Jacobson, Booch, và Rumbaugh, 1999] có hai mục tiêu tổng thể. Từ quan điểm của dòng công việc yêu cầu (dòng công việc trước đó), mục đích của dòng công việc phân tích là để hiểu sâu hơn về các yêu cầu. Nói cách khác, từ quan điểm của dòng công việc thiết kế và thực hiện (dòng công việc tuân theo dòng công việc phân tích), mục đích của dòng công việc phân tích là mô tả những yêu cầu đó theo cách mà thiết kế và triển khai kết quả dễ dàng duy trì .

Quy trình hợp nhất được định hướng theo trường hợp sử dụng. Trong quy trình phân tích, các trường hợp sử dụng được mô tả theo các lớp của sản phẩm phần mềm. Quy trình hợp nhất có ba loại lớp: lớp thực thể, lớp ranh giới và lớp điều khiển. Một lớp thực thể mô hình hóa thông tin tồn tại lâu dài. Trong trường hợp của một sản phẩm phần mềm ngân hàng, Lớp Tài khoản là một lớp thực thể vì thông tin trên các tài khoản phải nằm trong sản phẩm phần mềm. Đối với sản phẩm phần mềm MSG Foundation, Lớp Đầu tư là một lớp thực thể; một lần nữa, thông tin về các khoản đầu tư phải tồn tại lâu dài.

Một lớp ranh giới mô hình hóa sự tương tác giữa sản phẩm phần mềm và các tác nhân của nó. Các lớp ranh giới thường được liên kết với đầu vào và đầu ra. Ví dụ, trong sản phẩm phần mềm MSG Foundation, các báo cáo phải được in ra liệt kê các khoản đầu tư của Quỹ, cũng như tất cả các khoản thế chấp hiện đang nắm giữ. Điều này có nghĩa là cần có các lớp ranh giới Lớp Báo cáo Đầu tư và Lớp Báo cáo Thế chấp.

Một lớp điều khiển mô hình hóa các phép tính và thuật toán phức tạp. Trong trường hợp của sản phẩm phần mềm MSG Foundation, thuật toán ước tính quỹ khả dụng trong tuần là một lớp đối chứng, cụ thể là, Ước tính quỹ cho Lớp Tuần.

Kí hiệu UML cho ba loại lớp này được thể hiện trong Hình 13.1. Đây là những khuôn mẫu, tức là những phần mở rộng của UML. Một điểm mạnh của UML là nó cho phép xác định các cấu trúc bổ sung không phải là một phần của UML nhưng có thể cần thiết để mô hình hóa một hệ thống cụ thể một cách chính xác.

Như đã nêu ở phần đầu của phần này, trong quá trình phân tích công việc, các trường hợp sử dụng được mô tả theo các lớp của sản phẩm phần mềm. Bản thân Quy trình hợp nhất không mô tả cách trích xuất các lớp vì người dùng Quy trình hợp nhất được mong đợi có kiến ​​thức nền tảng về phân tích và thiết kế hướng đối tượng. Theo đó, cuộc thảo luận về Quy trình Hợp nhất này tạm thời bị tạm dừng để có thể đưa ra lời giải thích về cách các lớp được trích xuất; chúng ta quay lại Quy trình hợp nhất trong Phần 13.15.

Các lớp thực thể, tức là, các lớp mô hình hóa thông tin tồn tại lâu dài, được coi là đầu tiên.)

* 1. Extracting the Entity Classes (13.2 Trích xuất các lớp thực thể)

Entity class extraction consists of three steps that are carried out iteratively and incrementally:

* + 1. **Functional modeling**. Present scenarios of all the use cases (a **scenario** is an instance of a use case).
    2. **Entity class modeling**. Determine the entity classes and their attributes. Then, deter- mine the interrelationships and interactions between the entity classes. Present this information in the form of a class diagram.
    3. **Dynamic modeling**. Determine the operations performed by or on each entity class or subclass. Present this information in the form of a statechart.

However, as with all iterative and incremental processes, the three steps are not neces- sarily always performed in this order; a change in one model frequently triggers corre- sponding revisions of the other two models.

To show how this is done, we now extract the entity classes of the elevator problem case study.

(Dịch: Khai thác lớp thực thể bao gồm ba bước được thực hiện lặp đi lặp lại và tăng dần:

1. Mô hình chức năng. Trình bày các kịch bản của tất cả các ca sử dụng (một kịch bản là một thể hiện của ca sử dụng).

2. Mô hình lớp thực thể. Xác định các lớp thực thể và các thuộc tính của chúng. Sau đó, xác định các mối quan hệ và tương tác lẫn nhau giữa các lớp thực thể. Trình bày thông tin này dưới dạng biểu đồ lớp.

3. Mô hình hóa động. Xác định các hoạt động được thực hiện bởi hoặc trên mỗi lớp thực thể hoặc lớp con. Trình bày thông tin này dưới dạng một biểu đồ trạng thái.

Tuy nhiên, như với tất cả các quá trình lặp đi lặp lại và tăng dần, ba bước không bắt buộc phải luôn luôn được thực hiện theo thứ tự này; sự thay đổi trong một mô hình thường xuyên gây ra các bản sửa đổi tương ứng của hai mô hình còn lại.

Để chỉ ra cách thực hiện điều này, bây giờ chúng ta trích xuất các lớp thực thể của nghiên cứu điển hình về vấn đề thang máy.)

*ase Study*

*C*

Object-Oriented Analysis:

The Elevator Problem Case Study

The elevator problem case study is described in Chapter 12. For ease of reference, the problem is repeated here.

A product is to be installed to control n elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to the following constraints:

* + 1. Each elevator has a set of m buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited by the elevator.
    2. Each floor, except the first floor and the top floor, has two buttons, one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.
    3. When an elevator has no requests, it remains at its current floor with its doors closed. The first step in OOA is to model the use cases.

###### ase Study



*C*

**FIGURE 13.2**

Use-case diagram for the elevator problem case study.

Functional Modeling:

The Elevator Problem Case Study

A **use case** describes the interaction between the product to be constructed and the **actors**, that is, the external users of that product. The only interactions pos- sible between a user and an elevator are the user pressing an elevator button to summon an elevator or the user pressing a floor button to request the elevator to stop at a specific floor, hence, two use cases, Press an Elevator Button and Press a Floor Button. The two use cases are shown in the use-case diagram (Section 11.7) of Figure 13.2.

**User**

**FIGURE 13.3** The first iteration of a normal scenario (the missing responsibilities and the use of the passive voice will be corrected in the next iteration).

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 7.
2. The Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. The elevator doors open.
5. The timer starts.

User A enters the elevator.

1. User A presses the elevator button for floor 7.
2. The elevator button for floor 7 is turned on.
3. The elevator doors close after a timeout.
4. The Up floor button is turned off.
5. The elevator travels to floor 7.
6. The elevator button for floor 7 is turned off.
7. The elevator doors open to allow User A to exit from the elevator.
8. The timer starts.

User A exits from the elevator.

1. The elevator doors close after a timeout.
2. The elevator proceeds to floor 9 with User B.

A use case provides a generic description of the overall functionality; a scenario is a specific instantiation of a use case, just as an object is an instantiation of a class. In general, there are a large number of scenarios, each representing one specific set of interactions. In this section, we consider the scenario of Figure 13.3, which incor- porates instantiations of both use cases.

Figure 13.3 depicts a **normal scenario**; that is, a set of interactions between users and elevators that corresponds to the way we understand elevators should be used. Figure 13.3 was constructed after carefully observing different users interact- ing with elevators (or, more precisely, with elevator buttons and floor buttons). The 15 numbered events describe in detail the two interactions between User A and the buttons of the elevator system (event 1 and event 6) and the operations performed by the components of the elevator system (events 2 through 5 and 7 through 15). Two items, User A enters the elevator and User A exits from the elevator, are unnumbered. Such items essentially are comments; User A does not interact with the components of the elevator when entering or leaving an elevator.

In contrast, Figure 13.4 is an **exception scenario**. It depicts what happens when

a user presses the Up button at floor 3 but actually wants to go down to floor 1. This scenario, too, was constructed by observing the actions of many users in elevators; it is unlikely that someone who has never used an elevator would realize that users sometimes press the wrong button.

There is a serious mistake throughout Figures 13.3 and 13.4. Recall that, as stated in Section 1.9, **responsibility-driven design** is a feature of the object-oriented paradigm. From the very beginning of the life cycle, that is, from the requirements

**FIGURE 13.4** An exception scenario (the missing responsibilities and the use of the passive voice will be corrected in the next iteration).

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 1.
2. The Up floor button is turned on.
3. An elevator arrives at floor 3. It contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
4. The elevator doors open.
5. The timer starts.

User A enters the elevator.

1. User A presses the elevator button for floor 1.
2. The elevator button for floor 1 is turned on.
3. The elevator doors close after a timeout.
4. The Up floor button is turned off.
5. The elevator travels to floor 9.
6. The elevator button for floor 9 is turned off.
7. The elevator doors open to allow User B to exit from the elevator.
8. The timer starts.

User B exits from the elevator.

1. The elevator doors close after a timeout.
2. The elevator proceeds to floor 1 with User A.

workflow onward, it is essential to specify the responsibility for each action. Consider event 2 in Figure 13.3, The Up floor button is turned on. This statement does not specify who is responsible for turning on the button. Instead, the scenario should have stated, “The system turns on the Up floor button.” Similarly, event 4 states, The elevator doors open. But who or what is responsible for opening the doors? Is it a manual elevator in which the users have to open and close the doors? Or is it an auto- matic elevator in which the system is responsible for opening and closing the doors? Accordingly, in use cases and scenarios (instantiations of use cases), the responsibil- ity for each action must be explicitly stated.

Furthermore, it is bad practice to use the passive voice in a use case, a scenario, or in any other UML diagram that specifies actions. For example, event 2, The Up floor button is turned on, should not be in the passive voice. A use case describes an inter*action* between the software product and the user; for clarity, an action should be described in the active voice. Furthermore, a use case should be written from the user’s perspective, that is, what the user does and how the software product responds. Finally, it should be written in the present tense, to give a sense of immediacy.

In summary, statements in a use case or scenario should take the form, “A user does this and the software product responds by doing that.” In view of the fact that the use cases will eventually be refined into the run-time behavior of the product, statements in that form are easy to test, easy to document, and easy to modify. The mistakes in the scenarios of Figures 13.3 and 13.4 are corrected in a subsequent itera- tion, in Section 13.7.

###### ase Study

*C*

The scenarios of Figures 13.3 and 13.4, plus innumerable others, are specific instances of the use cases shown in Figure 13.2. The OOA team should study suf- ficient scenarios to gain a comprehensive insight into the behavior of the system being modeled. This information is used in the next step, entity class modeling, to determine the entity classes.

Entity Class Modeling:



The Elevator Problem Case Study

In this step, the entity classes and their attributes are extracted and represented in a UML class diagram (see Just in Case You Wanted to Know Box 13.2). Only the attributes of an entity class are determined at this time, not the methods; the latter are assigned to the classes during the object-oriented design (OOD) workflow.



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

**Just in Case You Wanted to Know**

**Box 13.2**

As explained at the beginning of Chapter 7, the object-oriented paradigm did not suddenly appear out of nowhere. Instead, it evolved out of the classical paradigm, in response to perceived shortcomings in the classical paradigm.

Entity class modeling is an example of this evolution. It is an extension of the classical technique of entity-relationship modeling. As described in Section 12.6, entity-relationship modeling has been used for database modeling since 1976.

A characteristic of the whole object-oriented paradigm is that the various steps rarely are easy to carry out. Fortunately, the benefits of using objects make the effort worthwhile. So it should not come as a surprise that the first part of the analysis workflow, extracting entity classes and their attributes, usually is difficult to get right the first time.

One method of determining the entity classes is to deduce them from the use cases. That is, the developers carefully study all the scenarios, both normal and exception, and identify the components that play a role in the use cases. From just the scenarios of Figures 13.3 and 13.4, candidate entity classes are elevator buttons, floor buttons, elevators, doors, and timers. As we will see, these candidate entity classes are close to the actual classes extracted during entity class modeling. In general, however, there are many scenarios and, consequently, a large number of potential classes. An inex- perienced developer may be tempted to infer too many candidate entity classes from the scenarios. This has a deleterious effect on the entity class modeling, because it is easier to add a new entity class than to remove a candidate entity class that should not have been included.

Another approach to determining the entity classes, which is effective when the developers have domain expertise, is CRC cards (Section 13.5.2). However, if the developers have little or no experience in the application domain, then it is advisable to use noun extraction, described in Section 13.5.1.

* + 1. Noun Extraction

For developers with no domain expertise, a good way to proceed is to use the fol- lowing two-stage **noun-extraction method** to extract candidate entity classes and then to refine the solution:

**Stage 1. Describe the Software Product in a Single Paragraph.**

One possible way to do this for the elevator problem case study is as follows:

Buttons in elevators and on the floors control the movement of n elevators in a build- ing with m floors. Buttons illuminate when pressed to request the elevator to stop at a specific floor; the illumination is canceled when the request has been satisfied. When an elevator has no requests, it remains at its current floor with its doors closed.

**Stage 2. Identify the Nouns.**

Identify the nouns in the informal strategy (excluding those that lie outside the problem boundary); then use these nouns as candidate entity classes. The informal strategy is now reproduced, but this time with the identified nouns printed in a sans serif typeface.

Buttons in elevators and on the floors control the movement of n elevators in a building with m floors. Buttons illuminate when pressed to request an elevator to stop at a specific floor; the illumination is canceled when the request has been satisfied.

When an elevator has no requests, it remains at its current floor with its doors closed.

There are eight different nouns: button, elevator, floor, movement, build- ing, illumination, request, and door. Three of these nouns—floor, building, and door—lie outside the problem boundary and therefore may be ignored. Three of the remaining nouns—movement, illumination, and request—are **abstract nouns**; that is, they identify things that have no physical existence. A useful rule of thumb is that abstract nouns rarely end up corresponding to classes. Instead, they frequently are attributes of classes. For example, illumination is an **attribute** of button.

This leaves two nouns and, therefore, two candidate entity classes: **Elevator Class** and **Button Class**. (The UML convention is to use boldface for class names and capitalize the initial letter of each word in a class name.)

The resulting **class diagram** is shown in Figure 13.5. **Button Class** has the Boolean attribute illuminated to model events 2, 7, 9, and 11 of the scenarios of Figures 13.3 and 13.4. The problem specifies two types of buttons, so two subclasses of **Button Class** are defined: **Elevator Button Class** and **Floor Button Class** (the open triangle denotes inheritance in UML). Each instance of **Elevator Button Class** and **Floor Button Class** communicates with the instance of **Elevator Class**. The latter class has the Boolean attribute doors open to model events 4, 8, 12, and 14 of the two scenarios.

Unfortunately, this is not a good beginning. In a real elevator, the buttons do not directly communicate with the elevators; some sort of elevator controller is needed, if only to decide which elevator to dispatch in response to a particular request. However, the problem statement makes no mention of a controller, so it was not selected as an entity class during the noun-extraction process. In other words, the technique of this section for finding candidate entity classes provides a starting point but certainly should not be relied on to do more than that.

**FIGURE 13.5**

The first iteration of the class diagram for the elevator problem case study.

m communicates

**Floor Button Class**

**Elevator Button Class**

|  |
| --- |
| **Button Class** |
| illuminated : Boolean |
|  |

with

1

doors open : Boolean

2m — 2

communicates with

n

**Elevator Class**

**FIGURE 13.6**

The second iteration of the class diagram for the elevator problem case study.

Adding the **Elevator Controller Class** to Figure 13.5 yields Figure 13.6. This certainly makes more sense. Furthermore, there are now one-to-many relation- ships in Figure 13.6, as opposed to the hard to model many-to-many relationship of Figure 13.5. It therefore seems reasonable to go on to stage 3 at this point, bearing in mind that it is possible to return to entity class modeling at any time, even as

|  |
| --- |
| **Button Class** |
| illuminated : Boolean |
|  |

mn

**Elevator Button Class**

2m — 2

**Floor Button Class**

controls controls

1

1

1

**Elevator Controller Class**

controls

n

|  |
| --- |
| **Elevator Class** |
| doors open : Boolean |
|  |





**Just in Case You Wanted to Know Box 13.3**

How do we find the number of days between February 21, 1999, and August 16, 2007? Such subtractions are needed in many financial computations, such as calculating an inter- est payment or determining the present value of a future cash flow. The usual way this is done is to convert each date into an integer, the number of days since a specified starting date. The problem is that we cannot agree what starting date to use.

Astronomers use Julian days, the number of days since noon GMT on January 1, 4713,

B.C.E. This system was invented in 1582 by Joseph Scaliger, who named it for his father, Julius Caesar Scaliger. (If you really, really have to know why January 1, 4713 B.C.E. was chosen, consult [USNO, 2000].)

A Lilian date is the number of days since October 15, 1582, the first day of the Gregorian calendar, introduced by Pope Gregory XIII. Lilian dates are named for Luigi Lilio, a leading proponent of the Gregorian calendar reform. Lilio was responsible for deriving many of the algorithms of the Gregorian calendar, including the rule for leap years.

Turning to software, COBOL intrinsic functions use January 1, 1600, as the starting date for integer dates. Almost all spreadsheets, however, use January 1, 1900, following the lead of Lotus 1-2-3.

late as the implementation workflow. However, before proceeding with the dynamic modeling, a different technique for entity class modeling is considered.

* + 1. CRC Cards

For a number of years, **class–responsibility–collaboration (CRC) cards** have been utilized during the object-oriented analysis workflow [Wirfs-Brock, Wilkerson, and Wiener, 1990]. For each class, the software development team fills in a card showing the name of the class, its functionality (responsibility), and a list of the other classes it invokes to achieve that functionality (collaboration).

This approach subsequently has been extended. First, a CRC card often explicitly contains the attributes and methods of the class, rather than just its “responsibility” expressed in some natural language. Second, the technology has changed. Instead of using cards, some organizations put the names of the classes on Post-it notes, which they move around on a white board; lines are drawn between the Post-it notes to denote collaboration. Nowadays the whole process can be automated; CASE tools like System Architect include components for creating and updating CRC “cards” on the screen.

The strength of CRC cards is that, when utilized by a team, the interaction among the members can highlight missing or incorrect fields in a class, whether attributes or methods. Also, the relationships between classes are clarified when CRC cards are used. One especially powerful technique is to distribute the cards among the team members, who then act out the responsibilities of their classes. Consequently, some- one might say, “I am the **Date Class**, and my responsibility is to create new date objects.” Another team member might then interject that he or she needs additional functionality from the **Date Class**, such as converting a date from the conven- tional format to an integer, the number of days from January 1, 1900, so that finding the number of days between any two dates can be computed easily by subtracting the corresponding two integers (see Just in Case You Wanted to Know Box 13.3). Accordingly, acting out the responsibilities of CRC cards is an effective means of verifying that the class diagram is complete and correct.

*ase Study*

*C*

A weakness of CRC cards is that this approach generally is not a good way of identify- ing entity classes unless the team members have considerable experience in the relevant application domain. On the other hand, once the developers have determined many of the classes and have a good idea of their responsibilities and collaborations, CRC cards can be an excellent way of completing the process and making sure that everything is correct. This is described in Section 13.7. First, however, we need to perform the dynamic modeling.

Dynamic Modeling:



The Elevator Problem Case Study

The aim of dynamic modeling is to produce a **statechart**, a description of the target product similar to a finite state machine, for each class. First, consider **Elevator Controller Class**. For simplicity, only one elevator is considered. The relevant statechart for **Elevator Controller Class** is in Figure 13.7.

The notation is somewhat similar to that of the finite state machine (FSM) of Section 12.7, but there is a significant difference. An FSM as presented in Chapter 12 is an example of a formal technique. The state transition diagrams themselves are not a complete representation of the product to be built. Instead, the model consists of a set of transition rules of the form given in equation (12.2):

**current state** and **event** and **predicate**  **next state**

Formality is achieved by presenting the model in the form of a set of mathematical rules.

In contrast, the representation of a UML statechart is somewhat less formal. The three aspects of a state machine (state, event, and predicate) are distributed over the UML diagram. For example, the state **Going Into Wait State** in Figure 13.7 is entered if the present state is **Elevator Event Loop** and the event elevator stopped, no requests pending is true. When the state **Going Into Wait State** has been entered, operation Close elevator doors after timeout is to be carried out. Current versions of OOA are semiformal (graphical) techniques, and the intrin- sic lack of formality of the statechart accordingly is no problem. However, when the object-oriented paradigm matures, it is likely that more formal versions will be developed and the corresponding dynamic models will be somewhat closer to finite state machines.

To see the equivalence of the statechart of Figure 13.7 and the STDs of Figures

12.15 through 12.17, consider various scenarios. For example, consider the first part of the scenario of Figure 13.3. Event 1 is User A presses the Up floor button at floor 3.

First consider the STD of Figure 12.16. If the floor button is off, then the button is turned on. Now consider the statechart of Figure 13.7. The solid circle denotes the start state, which takes the system into state **Elevator Event Loop**. Following the leftmost vertical line, if the button was turned off when it is pushed, the system enters

**FIGURE 13.7** The first iteration of the statechart for the **Elevator Controller Class**.



button pushed, button turned on

no requests pending, doors closed

button pushed, button turned off

elevator moving in direction d, floor f is next

elevator stopped, request(s) pending

elevator stopped, no requests pending,

**Determining If Stop Requested**

Check requests

no request user has

to stop at requested stop floor f at floor f

floor button floor

turned on button turned off

**Continuing Moving**

Move elevator one floor in direction d

**Stopping At Floor**

Stop elevator Open doors and start timer Update requests

**Turning Off Floor Button**

Turn off floor button

elevator button elevator turned on button

turned

**Turning Off Elevator Button**

Turn off elevator button

off

**Processing Next Request**

Move elevator one floor in direction of next request

Close elevator doors after timeout

**Closing Elevator Doors**

Close elevator doors after timeout

**Going Into Wait State**

**Elevator Event Loop**

|  |  |
| --- | --- |
| **Processing New Request** | |
| Turn on button Update requests | |
|  |  |

state **Processing New Request** of Figure 13.7, and the button is turned on. The following state is **Elevator Event Loop**.

Next, the elevator nears floor 3. First consider the STD approach. In Figure 12.17, the elevator goes into state S (U, 3); that is, it stops at floor 3, about to go up. (Because the simplifying assumption has been made of only one elevator, the argu- ment e in Figure 12.17 is suppressed here.) Now the doors close (Figure 12.17), the Up floor button is turned off (Figure 12.16), and the elevator starts to move toward floor 4.

Returning to the statechart of Figure 13.7, consider what happens when the elevator nears floor 3. Because the elevator is in motion, the next state entered is **Determining If Stop Requested**. The requests are checked and, because User A

has requested the elevator to stop there, the next state is **Stopping At Floor**. The elevator stops at floor 3, the doors open, and the timer starts. The elevator button for floor 3 has not been pressed, so state **Elevator Event Loop** is next.

User A enters and presses the elevator button for floor 7. Therefore, the next state is again **Processing New Request**, followed again by **Elevator Event Loop**. The elevator has stopped and two requests are pending, so state **Closing Elevator Doors** is next and the doors close after a timeout. The floor button at floor 3 was pressed by User A, so **Turning Off Floor Button** is the following state, and the floor button is turned off. State **Processing Next Request** is next, and the eleva- tor starts to move toward floor 4. The relevant aspects of the corresponding diagrams clearly are equivalent with respect to this scenario; you may wish to consider other possible scenarios as well.

From the preceding discussion, it should come as no surprise to learn that Figure

* 1. was constructed from the scenarios. More precisely, the specific events of the scenarios were generalized. For example, consider the first event of the scenario of Figure 13.3, User A presses the Up floor button at floor 3. This specific event is generalized to an arbitrary button (floor button or elevator button) being pushed. Then, there are two possibilities. Either the button already is turned on (in which case nothing happens) or the button is turned off (in which case action must be taken to process the user’s request).

To model this event, the **Elevator Event Loop** state is drawn in Figure 13.7. The case of an already turned on button is modeled by the do-nothing loop with event button pushed, button turned on in the top left-hand corner of Figure 13.7. The other case, a turned-off button, is modeled by the arrow labeled with the event button pushed, button turned off leading to state **Processing New Request**. From event 2 of the scenario it is clear that the operation Turn on button is needed in this state. Furthermore, the purpose of the user’s action of pressing an arbitrary button is to request an elevator (floor button) or request an elevator to move to a spe- cific floor (elevator button), so operation Update requests also must be carried out in the state **Processing New Request**.

Now consider event 3 of the scenario, An elevator arrives at floor 3. This was generalized to the concept of an arbitrary elevator moving between floors. The motion of the elevator is modeled by the event elevator moving in direction d, floor f is next and the state **Determining If Stop Requested**. But there again are two possibilities, either a request to stop at floor f or no such request. In the former case, corresponding to event no request to stop at floor f, the elevator simply must be in the state of **Continuing Moving** one more floor in direction d. In the latter case (corresponding to event user has requested stop at floor f), from the sce- nario of Figure 13.3 it is clear that it is necessary to Stop elevator (from event 3), and then Open doors and start timer (from events 4 and 5); state **Stopping At Floor** is needed to perform these actions. Also, similar to the **Processing New Request** state, it becomes apparent that it is necessary also to Update requests in state **Stopping At Floor**. In addition, generalizing event 9 of the scenario leads to the realization that the floor button has to be turned off if it is turned on. This is mod- eled by state **Turning Off Floor Button**, together with the two events above the box representing that state. Similarly, generalizing event 11 of the scenario implies

that the elevator button has to be turned off if it is turned on. This is modeled by state **Turning Off Elevator Button**, together with the two events above the box rep- resenting that state.

Generalizing event 8 of the scenario of Figure 13.3 yields state **Closing Elevator Doors**; generalizing event 10 yields state **Processing Next Request**. However, the need for the state **Going Into Wait State** and the event no requests pend- ing, doors closed is deduced by generalizing an event of a different scenario, one in which the user exits from the elevator but no buttons remain turned on.

* 1. **The Test Workflow: Object-Oriented Analysis**

At this point, the functional, entity class, and dynamic models appear to be complete and the **test workflow** resumes. The next step is to review the analysis workflow to date. One component of this review, as suggested in Section 13.5.2, is to use CRC cards.

Accordingly, CRC cards are filled in for each of the entity classes, **Button Class**, **Elevator Button Class**, **Floor Button Class**, **Elevator Class**, and **Elevator Controller Class**. The CRC card for **Elevator Controller Class**, shown in Figure 13.8, is deduced from the class diagram of Figure 13.5 and the statechart of Figure 13.6. In more detail, the RESPONSIBILITY of **Elevator Controller Class** is obtained by listing all the operations in the statechart for **Elevator Controller Class** (Figure 13.7). The COLLABORATION of the **Elevator Controller Class** is determined by examining the class diagram of Figure 13.6 and noting that classes **Elevator Button Class**, **Floor Button Class**, and **Elevator Class** interact with class **Elevator Controller Class**.

**FIGURE 13.8**

|  |
| --- |
| CLASS  **Elevator Controller Class** |
| RESPONSIBILITY   1. Turn on elevator button 2. Turn off elevator button 3. Turn on floor button 4. Turn off floor button 5. Move elevator up one floor 6. Move elevator down one floor 7. Open elevator doors and start timer 8. Close elevator doors after timeout 9. Check requests 10. Update requests |
| COLLABORATION   1. **Elevator Button Class** 2. **Floor Button Class** 3. **Elevator Class** |

The first iteration of the CRC card for the **Elevator Controller Class**.

This CRC card highlights two major problems with the first iteration of the object- oriented analysis.

1. Consider responsibility 1. Turn on elevator button. This command is totally out of place in the object-oriented paradigm. From the viewpoint of responsibility-driven design (Section 1.9), objects (instances) of **Elevator Button Class** are respon- sible for turning themselves on or off. Also, from the viewpoint of information hiding (Section 7.6), the **Elevator Controller Class** should not have the knowledge of the internals of **Elevator Button Class** needed to turn on a button. The correct responsibility is this: Send a message to **Elevator Button Class** to turn itself on. Similar changes are needed for responsibilities 2 through 6 in Figure 13.8. These six corrections are reflected in Figure 13.9, the second iteration of the CRC card for the **Elevator Controller Class**.
2. A class has been overlooked. Returning to Figure 13.8, consider responsibility 7. Open elevator doors and start timer. The key concept here is the notion of **state**. The attri- butes of a class sometimes are termed **state variables**. The reason for this terminology is that, in most object-oriented implementations, the state of the product is determined by the values of the attributes of the various component objects. The statechart has many features in common with a finite state machine. Accordingly, it is not surprising that the concept of state plays an important role in the object-oriented paradigm. This concept can be used to help determine whether a component should be modeled as a class. If the component in question possesses a state that is changed during execution of the implementation, then it probably should be modeled as a class. Clearly, the doors of the elevator possess a state (open or closed), and **Elevator Doors Class** therefore should be a class.

**FIGURE 13.9**

|  |
| --- |
| CLASS  **Elevator Controller Class** |
| RESPONSIBILITY   1. Send message to **Elevator Button Class** to turn on button 2. Send message to **Elevator Button Class** to turn off button 3. Send message to **Floor Button Class** to turn on button 4. Send message to **Floor Button Class** to turn off button 5. Send message to **Elevator Class** to move up one floor 6. Send message to **Elevator Class** to move down one floor 7. Send message to **Elevator Doors Class** to open 8. Start timer 9. Send message to **Elevator Doors Class** to close after timeout 10. Check requests 11. Update requests |
| COLLABORATION   1. **Elevator Button Class** (subclass) 2. **Floor Button Class** (subclass) 3. **Elevator Doors Class** 4. **Elevator Class** |

The second iteration of the CRC card for the **Elevator Controller Class**.

Chapter 13 *Object-Oriented Analysis* **419**



**Just in Case You Wanted to Know**

**Box 13.4**

Some years ago, I was on the 10th floor of a building, waiting impatiently for an elevator. The doors opened, I started to step forward—only no elevator was there. What saved my life was the total blackness I saw as I was about to step into the elevator shaft, and I instinc- tively realized that something was wrong.

Perhaps, if that elevator control system had been developed using the object-oriented paradigm, the inappropriate opening of the doors on the 10th floor might have been avoided.

There is another reason why **Elevator Doors Class** should be a class. The object- oriented paradigm allows the state to be hidden within an object and hence protected from unauthorized change. If there is an **Elevator Doors Class** object, the only way that the doors of the elevator can be opened or shut is by sending a message to that **Elevator Doors Class** object. Serious accidents can be caused by opening or closing the doors of an elevator at the wrong time; see Just in Case You Wanted to Know Box 13.4. Therefore, for certain types of products, safety considerations should be added to the other strengths of objects listed in Chapters 7 and 8.

Adding **Elevator Doors Class** means that responsibilities 7 and 8 in Figure 13.8

need to be changed analogously to responsibilities 1 through 6. That is, messages should be sent to instances of the **Elevator Doors Class** to open and close themselves. But there is an additional complication.

Recall that responsibility 7 is Open elevator doors and start timer. This must be split into two separate responsibilities. A message must indeed be sent to **Elevator Doors Class** to open. However, the timer is part of the **Elevator Controller Class**, and start- ing the timer therefore is the responsibility of the **Elevator Controller Class** itself. The second iteration of the CRC card for **Elevator Controller Class** (Figure 13.9) shows that this separation of responsibilities has been achieved satisfactorily.

In addition to the two major problems highlighted by the CRC card of Figure 13.8, responsibilities Check requests and Update requests of **Elevator Controller Class** require the attribute requests be added to **Elevator Controller Class**. At this stage, requests are defined simply to be of type requestType; a data structure for requests will be chosen during the design workflow.

The corrected class diagram is shown in Figure 13.10. Having modified the class diagram, we must reexamine the use-case diagram and statecharts to see if they, too, need further refinement. The use-case diagram clearly is still adequate. However, the operations in the statechart of Figure 13.7 must be modified to reflect the responsibili- ties of Figure 13.9 (the second iteration of the CRC card) and not Figure 13.8 (the first iteration). Also, the set of statecharts must be extended to include the additional class. The scenarios need to be updated to reflect these changes; Figure 13.11 shows the second iteration of the scenario of Figure 13.3.

There is a serious problem in Figure 13.10, the third iteration of the class diagram. The **Elevator Controller Class** is running the entire show—this is an example of a so- called God class, a class that is exposed to too much information and has too much control. This type of architecture is a well-known antipattern, or pattern to be avoided (see Just in Case You Wanted to Know Box 8.4). To solve this problem, instead of having one cen- tral elevator controller, we distribute the control. Each of the n elevators now has its own

**FIGURE 13.10**

|  |
| --- |
| **Button Class** |
| illuminated : Boolean |
|  |

The third iteration of the class diagram for the elevator problem case study.

mn

**Elevator Button Class**

2m — 2

**Floor Button Class**

controls controls

1

requests : requestType

**Elevator Controller Class**

1

controls

n

1

1controlsn

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

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

elevator subcontroller, and each of the m floors has its own floor subcontroller. The m + n subcontrollers all communicate with a scheduler, which processes requests. The resulting fourth iteration of the class diagram is shown in Figure 13.12. This diagram reflects a dis- tributed, decentralized architecture, characteristic of the object-oriented paradigm.

Now, when a user presses a **Floor Button Class** object, the **Floor Button Class** object sends a message to the corresponding **Floor Subcontroller Class** object inform- ing it that the button has been pressed. The **Floor Subcontroller Class** object sends a message back to the **Floor Button Class** object to ask whether its light is on. If not, it sends a message to that **Floor Button Class** object to turn itself on, and it also informs the **Scheduler Class** object of the new request that has been made by a user.

Similarly, when a user presses an **Elevator Button Class** object, the **Elevator Button Class** object sends a message to the corresponding **Elevator Subcontroller Class** object informing it that the button has been pressed. The **Elevator Subcon- troller Class** object sends a message back to the **Elevator Button Class** object to ask whether its light is on. If not, it sends a message to that **Elevator Button Class** object to turn itself on, and it also informs the **Scheduler Class** object of the new request that has been made.

Now, there is a sensor just above and just below each floor in each elevator shaft, for a total of 2m – 2 sensors per shaft. When an **Elevator Class** object nears a floor (moving up or down), the corresponding **Sensor Class** object sends an appropriate message to the corresponding **Elevator Subcontroller Class** object. The **Elevator Subcontroller Class** object then sends a message to the **Scheduler Class** object informing it that the

Chapter 13 *Object-Oriented Analysis* **421**

**FIGURE 13.11** The second iteration of a normal scenario for the elevator problem case study.

1. User A presses the Up floor button at floor 3 to request an elevator. User A wishes to go to floor 7.
2. The floor button informs the elevator controller that the floor button has been pushed.
3. The elevator controller sends a message to the Up floor button to turn itself on.
4. The elevator controller sends a series of messages to the elevator to move itself up to floor 3. The elevator contains User B, who has entered the elevator at floor 1 and pressed the elevator button for floor 9.
5. The elevator controller sends a message to the elevator doors to open themselves.
6. The elevator controller starts the timer. User A enters the elevator.
7. User A presses elevator button for floor 7.
8. The elevator button informs the elevator controller that the elevator button has been pushed.
9. The elevator controller sends a message to the elevator button for floor 7 to turn itself on.
10. The elevator controller sends a message to the elevator doors to close themselves after a timeout.
11. The elevator controller sends a message to the Up floor button to turn itself off.
12. The elevator controller sends a series of messages to the elevator to move itself up to floor 7.
13. The elevator controller sends a message to the elevator button for floor 7 to turn itself off.
14. The elevator controller sends a message to the elevator doors to open themselves to allow User A to exit from the elevator.
15. The elevator controller starts the timer. User A exits from the elevator.
16. The elevator controller sends a message to the elevator doors to close themselves after a timeout.
17. The elevator controller sends a series of messages to the elevator to move itself up to floor 9 with User B.

**Elevator Class** object is nearing that floor. The **Scheduler Class** object now checks whether there is a request to stop at that floor. If not, it sends a message to the **Elevator Subcontroller Class** object, which then sends a message to the appropriate **Elevator Class** object to move itself one further floor in the same direction. But if there is a request to stop, the **Scheduler Class** object informs the **Elevator Subcontroller Class** object accordingly, and then updates its request list appropriately. The **Elevator Subcontroller Class** object then sends a message to the relevant **Elevator Button Class** object to ask whether its light is off. If not, it sends a subsequent message to that **Elevator Button Class** object to turn itself off.

When an **Elevator Class** object stops at a floor, the corresponding **Elevator Sub- controller Class** object sends a message to the appropriate **Elevator Doors Class** object to open itself; it then starts its timer. After a time-out, it sends the appropriate mes- sage to that **Elevator Doors Class** object to close itself.

**FIGURE 13.12** The fourth iteration of the class diagram for the elevator problem case study.

1..2

m

controls

controls

1

**Floor Subcontroller Class**

communicates with

1

**Elevator Subcontroller Class**

2m — 2

communicates

with

1

1

1

controls 1

m communicates

with

n communicates

with

controls

1

1

1

requests: requestType

**Elevator Class**

**Scheduler Class**

**Sensor Class**

**tton Class**

**Elevator Bu**

**on Class**

**Floor Butt**

|  |
| --- |
| **Button Class** |
| illuminated : Boolean |
|  |

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

Finally, when an **Elevator Class** object leaves a floor (moving up or down), the appro- priate **Sensor Class** object informs the corresponding **Elevator Subcontroller Class** object that the elevator has left the floor. The **Elevator Subcontroller Class** object sends a message to the corresponding **Floor Subcontroller Class** object informing it that the elevator has left that floor, and the direction in which it is moving. The **Floor Subcon- troller Class** object then sends a message to the corresponding **Floor Button Class** object to determine if its light is on and, if so, sends a subsequent message to turn itself off.

The various UML diagrams now need to be updated to reflect the fourth iteration of the class diagram of Figure 13.12. The first iteration of the statechart for the **Elevator Subcontroller Class** is shown in Figure 13.13. The first iteration of the CRC card for

**FIGURE 13.13** The first iteration of the statechart for the **Elevator Subcontroller Class**.

no requests pending, doors closed

elevator button pushed

**Elevator Subcontroller Event Loop**

elevator moving in direction d, floor f is next

elevator stopped, request(s) pending

elevator stopped, no requests pending

**Checking If Elevator Button Is Turned On 1**

Send message to elevator button

**Waiting For Sensor Message**

**Going Into Wait State**

Close elevator doors

elevator button turned off

elevator button turned on

sensor message received

after timeout

**Determining If Stop Requested**

Send message to scheduler that elevator is nearing floor f

**Processing Next Request**

Send message to elevator doors to close after timeout

no request

to stop at floor f

user has

requested stop at floor f

Send message to elevator to move one floor in direction d

Wait for sensor message

**Processing New Request**

Send message to elevator button to turn itself on

Send message to scheduler that

**Continuing Moving**

Send message to elevator to move itself one

**Checking If Elevator Button Is Turned On 2**

Send message to elevator button

a new request has been made

further floor in direction d

elevator button

turned off

elevator button

turned on

sensor message

received

**Processing Existing Request**

Send message to elevator button to turn itself off

**Updating Requests, Buttons**

Send message to floor subcontroller that elevator has left floor moving in direction d



**423**

|  |  |
| --- | --- |
| **Stopping At Floor** | |
| Send message to doors to open Start timer | |
|  |  |

**FIGURE 13.14**

The first iteration of the CRC card for the **Elevator Sub- controller Class**.

|  |
| --- |
| CLASS  **Elevator Subcontroller Class** |
| RESPONSIBILITY   1. Send message to **Elevator Button Class** to check if it is turned on 2. Send message to **Elevator Button Class** to turn itself on 3. Send message to **Elevator Button Class** to turn itself off 4. Send message to **Elevator Doors Class** to open themselves 5. Start timer 6. Send message to **Elevator Doors Class** to close themselves after timeout 7. Send message to **Elevator Class** to move itself up one floor 8. Send message to **Elevator Class** to move itself down one floor 9. Send message to **Scheduler Class** that a request has been made 10. Send message to **Scheduler Class** that a request has been satisfied 11. Send message to **Scheduler Class** to check if the elevator is to stop at the next floor 12. Send message to **Floor Subcontroller Class** that elevator has left floor |
| COLLABORATION   1. **Elevator Button Class** (subclass) 2. **Sensor Class** 3. **Elevator Doors Class** 4. **Elevator Class** 5. **Scheduler Class** 6. **Floor Subcontroller Class** |

the **Elevator Subcontroller Class** is shown in Figure 13.14. Updating the other UML diagrams is left as an exercise (Problems 13.1–13.5).

Even after all these changes have been made and checked (including the modified CRC cards), it still may be necessary during the object-oriented design workflow to return to the object-oriented analysis workflow and revise one or more of the analysis artifacts. How- ever, at this stage it appears that the entity classes for the elevator problem case study have been correctly extracted.

* 1. Extracting the Boundary and Control Classes

Unlike entity classes, boundary classes are usually easy to extract. In general, each input screen, output screen, and printed report is modeled by its own boundary class. Recall that a class incorporates attributes (data) and operations. The boundary class modeling (say) a printed report incorporates all the various data items that can be included in the report and the various operations carried out to print the report.

Control classes are usually as easy to extract as boundary classes. In general, each non- trivial computation is modeled by a control class.

**FIGURE 13.15**

The seventh iteration of the use-case diagram of the MSG

Foundation case study.

**MSG Foundation Information System**

Estimate Funds Available for Week

Manage an Investment

Manage a Mortgage

**MSG Staff Member**

Update Estimated

Annual Operating Expenses

**Borrowers**

Produce a Report

###### ase Study

*C*

We now illustrate entity, boundary, and control class extraction and obtain further insights into the Unified Process by extracting the classes of the MSG Foundation case study. The starting point is the use-case diagram of Figure 11.42, reproduced here as Figure 13.15.

The Initial Functional Model:



The MSG Foundation Case Study

As described in Section 13.2, functional modeling consists of finding the scenarios of the use cases. Recall that a scenario is an instance of a use case. Consider the use case Manage a Mortgage (Figures 11.32 and 11.33). One possible scenario is shown in Figure 13.16. There is a change in the annual real-estate tax to be paid on a home for which the MSG Foundation has provided a mortgage. Because the bor- rowers pay this tax in equal weekly payments, any change in the real-estate tax must be entered in the relevant mortgage record, so that the total weekly installment (and perhaps the grant) can be adjusted accordingly. The normal portion of the extended scenario models an MSG staff member accessing the relevant mortgage record and

**FIGURE 13.16** An extended scenario of managing a mortgage.

An MSG Foundation staff member wants to update the annual real-estate tax on a home for which the Foundation has provided a mortgage.

1. The staff member enters the new value of the annual real-estate tax.
2. The information system updates the date on which the annual real-estate tax was last changed.

**Possible Alternative**

A. The staff member enters the mortgage number incorrectly.

**FIGURE 13.17** Another extended scenario of managing a mortgage.

There is a change in the weekly income of a couple who have borrowed money from the MSG Foundation. They wish to have their weekly income updated

in the Foundation records by an MSG staff member so that their mortgage payments will be correctly computed.

1. The staff member enters the new value of the weekly income.
2. The information system updates the date on which the weekly income was last changed.

**Possible Alternatives**

* 1. The staff member enters the mortgage number incorrectly.
  2. The borrowers do not bring documentation regarding their new income.

changing the annual real-estate tax. Sometimes, however, the staff member may not be able to locate the correct mortgage stored in the software product because he or she has entered the mortgage number incorrectly. This possibility is modeled by the exception portion of the scenario.

A second scenario corresponding to the Manage a Mortgage use case (Figures

11.32 and 11.33) is shown in Figure 13.17. Here the borrowers’ weekly income has changed. They would like this information to be reflected in the MSG Foundation records so that their weekly installment can be correctly computed. The normal portion of this extended scenario shows this operation proceeding as expected. The abnormal portion of this scenario shows two possibilities. First, as in the previous scenario, the staff member may enter the mortgage number incorrectly. Second, the borrowers may not bring with them adequate documentation to support their claim regarding their income, in which case the requested change is not implemented.

A third scenario (Figure 13.18) is an instance of use case Estimate Funds Available for Week (Figure 11.42). This scenario is directly derived from the description of the use case (Figure 11.43).

The scenarios of Figures 13.19 and 13.20 are instances of use case Produce a Report. Again, these scenarios are directly derived from the corresponding description of the use case (Figure 11.39). The remaining scenarios are equally straightforward and are therefore left as an exercise (Problems 13.12 and 13.13).

**FIGURE 13.18** A scenario of the Estimate Funds Available for Week use case.

|  |
| --- |
| An MSG Foundation staff member wishes to determine the funds available for mortgages this week.   1. For each investment, the information system extracts the estimated annual return on that investment. It sums the separate returns and divides the result by 52 to yield the estimated investment income for the week. 2. The information system then extracts the estimated annual MSG Foundation operating expenses and divides the result by 52. 3. For each mortgage:    1. The information system computes the amount to be paid this week by adding the principal and interest payment to 1\_nd of the sum of the annual real-estate   52  tax and the annual homeowner’s insurance premium.   * 1. It then computes 28 percent of the couple’s current gross weekly income.   2. If the result of Step 3.1 is greater than the result of Step 3.2, then it determines the mortgage payment for the week as the result of Step 3.2, and the amount of the grant for this week as the difference between the result of Step 3.1 and the result of Step 3.2.   3. Otherwise, it takes the mortgage payment for this week as the result of Step 3.1, and there is no grant for the week.  1. The information system sums the mortgage payments of Steps 3.3 and 3.4 to yield the estimated total mortgage payments for the week. 2. It sums the grant payments of Step 3.3 to yield the estimated total grant payments for the week. 3. The information system adds the results of Steps 1 and 4 and subtracts the results of Steps 2 and 5. This is the total amount available for mortgages for the current week. 4. Finally, the software product prints the total amount available for new mortgages during the current week. |

**FIGURE 13.19** A scenario of the Produce a Report

use case.

An MSG staff member wishes to print a list of all mortgages.

1. The staff member requests a report listing all mortgages.

**FIGURE 13.20** Another scenario of the Produce a Report use case.

An MSG staff member wishes to print a list of all investments.

1. The staff member requests a report listing all investments.

###### ase Study

*C*

13.10

The Initial Class Diagram:

The MSG Foundation Case Study

The second step is class modeling. The aim of this step is to extract the entity classes, determine their interrelationships, and find their attributes. The best way to start this step is usually to use the two-stage noun extraction method (Section 13.5.1).

In Stage 1 we describe the software product in a single paragraph. In the case of the MSG Foundation case study, a way to do this is

Weekly reports are to be printed showing how much money is available for mortgages. In addition, lists of investments and mortgages must be printed on demand.

In Stage 2 we identify the nouns in this paragraph. For clarity, the nouns are printed in sans serif type.

Weekly reports are to be printed showing how much money is available for mortgages. In addition, lists of investments and mortgages must be printed on demand.

The nouns are report, money, mortgage, list, and investment. Nouns report and list are not long lived, so they are unlikely to be entity classes (report will surely turn out to be a boundary class), and money is an abstract noun. This leaves two candidate entity classes, namely, **Mortgage Class** and **Investment Class**, as shown in Figure 13.21, the first iteration of the class diagram.

Now we consider interactions between these two entity classes. Looking at the descriptions of use cases Manage an Investment and Manage a Mortgage (Figures 11.31 and 11.33, respectively) it appears that the operations performed on the two entity classes are likely to be very similar, namely, insertions, deletions, and modifications. Also, the second iteration of the description of use case Produce a Report (Figure 11.39) shows all the members of both entity classes have to be printed on demand. In other words, **Mortgage Class** and **Investment Class** should probably be subclasses of some superclass. We will call that superclass **Asset Class**, because mortgages and investments are both assets of the MSG Foundation. The resulting second iteration of the class diagram is shown in Figure 13.22.



**FIGURE 13.21** The first iteration of the class diagram of the MSG Foundation case study.

**Mortgage Class**

**Investment Class**

**FIGURE 13.22**

**Mortgage Class**

**Investment Class**

**Asset Class**

The second iteration of the class diagram of the MSG Foundation case study.

**FIGURE 13.23**

The eighth iteration of the use-case diagram of the MSG Foundation case study. The new use case, Manage

an Asset, is shaded.

**MSG Staff Member**

**MSG Foundation Information System**

Estimate Funds Available for Week

Manage an Asset

Update Estimated

Annual Operating Expenses

Produce a Report

**Borrowers**



A useful side effect of constructing this superclass is that we can once again reduce the number of use cases. As shown in Figure 13.15, we currently have five use cases, including Manage a Mortgage and Manage an Investment. However, if we consider a mortgage or an investment to be a special case of an asset, we can combine the two use cases into a single use case, Manage an Asset. The eighth iteration of the use-case diagram is shown in Figure 13.23. The new use case is shaded. Now the attributes are added, as shown in Figure 13.24.

The phrase “iteration and *in*crementation” also includes the possibility of the need for a *de*crementation in what has been developed to date. There are two reasons for

**FIGURE 13.24** Attributes added to the second iteration of the class diagram of the MSG Foundation case study.

|  |
| --- |
| **Asset Class** |
| assetNumber |
|  |

**Investment Class**

investmentName estimatedAnnualReturn dateEstimatedReturnUpdated

**Mortgage Class**

lastNameOfMortgagees originalPurchasePrice dateMortgageIssued weeklyPrincipalAndlnterestPayment combinedWeeklyIncome mortgageBalance dateCombinedWeeklyIncomeUpdated annualRealEstateTax dateAnnualRealEstateTaxUpdated annualInsurancePremium dateAnnualInsurancePremiumUpdated

###### ase Study

*C*

13.11

such a decrease. First, if a mistake is made, the best way to correct it may be to **backtrack** to an earlier version of the software product and find a better way of performing the step that was incorrectly carried out. When backtracking, everything that was added in the course of the incorrect step now has to be removed. Second, as a consequence of reorganizing the models to date, one or more artifacts may have become superfluous. Developing a software product is hard. It is therefore important to remove superfluous use cases or other artifacts as soon as possible.

The Initial Dynamic Model:



The MSG Foundation Case Study

The third step in object-oriented analysis is dynamic modeling. In this step, a state- chart is drawn that reflects all the operations performed by or to that system, indi- cating the events that cause the transition from state to state. The major source of information regarding the relevant operations is the scenarios.

The statechart of Figure 13.25 reflects the operations of the complete MSG Foundation case study. The solid circle on the top left represents the initial state, the starting point of the statechart. The arrow from the initial state leads us to the state labeled **MSG Foundation Event Loop**; states other than the initial and final states

**FIGURE 13.25** The initial statechart of the MSG Foundation case study.



quit selected

estimate funds for the week selected

manage an asset selected

update estimated annual operating expenses selected

produce a report selected

**MSG Foundation Event Loop**

|  |  |
| --- | --- |
| **Estimating Funds For The Week** | |
| Estimate and print funds available for the current week | |
|  |  |

|  |  |
| --- | --- |
| **Managing An Asset** | |
| Add, delete, or modify a mortgage or investment | |
|  |  |

|  |  |
| --- | --- |
| **Updating Estimated Annual Operating Expenses** | |
| Update the estimated annual operating expenses | |
|  |  |

|  |  |
| --- | --- |
| **Producing A Report** | |
| Print a list of all mortgages or investments | |
|  |  |

are represented by rectangles with rounded corners. In state **MSG Foundation Event Loop**, one of five events can occur. In more detail, an MSG staff member can issue one of five commands: estimate funds for the week, manage an asset, update esti- mated annual operating expenses, produce a report, or quit. These possibilities are indi- cated by the five events estimate funds for the week selected, manage an asset selected, update estimated annual operating expenses selected, produce a report selected, and quit selected. (An **event** causes a **transition** between states.) When the system is in state **MSG Foundation Event Loop**, any one of the five events may occur, depending on which option the MSG staff member selects from the menu, shown in Figure 13.26, that will be incorporated in the target software prod- uct. [The C++ and Java implementations of the MSG Foundation case study given in Appendices H and I, respectively, use a textual interface rather than a graphical user interface (GUI). That is, instead of clicking on a box, as shown in Figure 13.26, the user types in a choice, as shown in Figure 13.27. For example, the user types 1 to Estimate funds available for week, 2 to Manage an asset, and so on. The reason the implementations in Appendices H and I use a textual interface, such as Figure 13.27, is that a textual interface can be run on all computers; a GUI generally needs

special software.]

Suppose that the MSG staff member clicks on the choice *Manage an asset* in the menu of Figure 13.26. The event manage an asset selected (second from the left below the **MSG Foundation Event Loop** box in Figure 13.25) has now occurred, so the system moves from its current state, **MSG Foundation Event Loop**, to the state **Managing An Asset**. The operations that the MSG staff member can perform in this state, namely, Add, delete, or modify a mortgage or investment, appear below the line in the box with rounded corners.

**FIGURE 13.26** Menu in the target MSG Foundation case study.

**FIGURE 13.27** Textual version of the menu of Figure 13.26.



*Quit*

Click on your choice: *Estimate funds for the week Manage an asset*

*Update estimated annual operating expenses Produce a report*

MAIN MENU

MARTHA STOCKTON GREENGAGE FOUNDATION

1. Estimate funds available for week
2. Manage an asset
3. Update estimated annual operating expenses
4. Produce a report
5. Quit

Type your choice and press <ENTER>:

###### ase Study

*C*

13.12

Once the operation has been performed, the system returns to the state **MSG Foundation Event Loop**, as shown by the arrows. The behavior of the rest of the statechart is equally straightforward.

In summary, the software product moves from state to state. In each state, the MSG staff member can perform the operations supported by that state, as listed below the line in the box with rounded corners that represents the state. This continues until the MSG staff member clicks on menu choice *Quit* when the soft- ware product is in the state **MSG Foundation Event Loop**. At this time the software product enters the final state (represented by the white circle containing the small black circle). When this state is entered, execution of the statechart ter- minates; recall that the statechart is a model of the execution of the target software product.

Revising the Entity Classes:



The MSG Foundation Case Study

The initial functional model, the initial class diagram, and the initial dynamic model have now been completed. However, a check of all three models reveals that some- thing has been overlooked.

Look at the initial statechart of Figure 13.25 and consider state **Updating Estimated Annual Operating Expenses** with operation Update the esti-

mated annual operating expenses. This operation has to be performed on data, namely, the current value of the estimated annual operating expenses. But where is the value of the estimated annual operating expenses to be found? Looking at Figure 13.24, it would have been a serious error to have it as an attribute of **Asset Class** or either of its subclasses. On the other hand, currently there is only one class **Asset Class**) and its two subclasses. This means that the only way a value

can be stored on a long-term basis is as an attribute of an instance of that class or its subclasses.

The solution is obvious: Another entity class is needed in which the value of the estimated annual operating expenses can be stored. In fact, other values need to be stored as well; the result is shown in Figure 13.28. A new class, **MSG Application Class**, has been introduced in which the various attributes shown in the top box in the figure can be stored. In addition, the **MSG Application Class** will be assigned the task of starting the execution of the rest of the software product.

Now the class diagram of Figure 13.28 is redrawn to reflect the stereotypes. This is shown in Figure 13.29. All four classes are entity classes. The entity classes seem to be correct, at least for now. The next step is to determine the boundary classes and control classes.

**FIGURE 13.28** The third iteration of the class diagram of the MSG Foundation case study.

|  |
| --- |
| **MSG Application Class** |
| estimatedAnnualOperatingExpenses dateEstimatedAnnualOperatingExpensesUpdated availableFundsForWeek expectedAnnualReturnOnInvestments dateExpectedAnnualReturnOnInvestmentsUpdated expectedGrantsForWeek expectedMortgagePaymentsForWeek |
|  |

|  |
| --- |
| **Asset Class** |
| assetNumber |
|  |

**Investment Class**

investmentName estimatedAnnualReturn dateEstimatedReturnUpdated

**Mortgage Class**

lastNameOfMortgagees originalPurchasePrice dateMortgageIssued weeklyPrincipalAndlnterestPayment combinedWeeklyIncome mortgageBalance dateCombinedWeeklyIncomeUpdated annualRealEstateTax dateAnnualRealEstateTaxUpdated annualInsurancePremium dateAnnualInsurancePremiumUpdated

**FIGURE 13.29**

Figure 13.28 redrawn to show the stereotypes.

**MSG**

**Application Class**

**Asset Class**

###### Case Study



**Investment Class**

**Mortgage Class**

13.13

**FIGURE 13.30**

The initial boundary classes of the MSG Foundation case study.

Extracting the Boundary Classes:

The MSG Foundation Case Study

Extracting entity classes is usually considerably harder than extracting boundary classes. After all, entity classes generally have interrelationships, whereas each input screen, output screen, and printed report is usually modeled by an (independent) boundary class, as pointed out in Section 13.8.

In view of the fact that the target MSG Foundation software product appears to be relatively straightforward (at least at this early stage of the Unified Process), it is rea- sonable to try to have just one screen that the MSG staff member can use for all four use cases: Estimate Funds Available for Week, Manage an Asset, Update Estimated Annual Operating Expenses, and Produce a Report. As more is learned about the MSG Foundation, it is certainly possible that this one screen may have to be refined into two or more screens. But the initial class extraction has just the one screen class, **User Interface Class**.

There are three reports that have to be printed, the estimated funds for the week report and the two asset reports, namely, the complete listing of all mortgages or of all investments. Each of these has to be modeled by a separate boundary class because the content of each report is different. The four corresponding initial bound- ary classes are then **User Interface Class**, **Estimated Funds Report Class**, **Mortgages Report Class**, and **Investments Report Class**. These four classes are displayed in Figure 13.30.

**User Interface Class Estimated Funds Report Class Mortgages Report Class Investments Report Class**

###### ase Study

*C*

13.14

**FIGURE 13.31** The initial control

class of the MSG Foundation case study.

|  |
| --- |
| **Estimate Funds for Week Class** |

Extracting the Control Classes: The MSG Foundation Case Study



Control classes are generally as easy to extract as boundary classes because each nontrivial computation is almost always modeled by a control class, as stated in Section 13.8. For the MSG Foundation case study, there is just one computation, namely, estimating the funds available for the week. This yields the initial control class **Estimate Funds for Week Class** shown in Figure 13.31.

The next step is to check all three sets of classes: entity classes, boundary classes, and control classes. Careful examination of the classes yields no obvious discrepan- cies. Having completed class extraction, we now return to the Unified Process.

###### ase Study



*C*

13.15

Use-Case Realization:

The MSG Foundation Case Study

A use case is a description of an interaction between an actor and the software product. Use cases are first utilized at the beginning of the software life cycle, that is, in the requirements workflow. During the analysis and design workflows, more details are added to each use case, including a description of the classes involved in carrying out the use case. This process of extending and refining use cases is called **use-case realization**. Finally, during the implementation workflow, the use cases are implemented in code.

This terminology is somewhat confusing, because the verb *realize* can be used in at least three different senses:

* Understand (“Harvey slowly began to realize that he was in the wrong classroom”).
* Receive (“Ingrid will realize a profit of $45,000 on the stock transaction”).
* Accomplish (“Janet hopes to realize her dream of starting a software development organization”).

In the phrase *realize a use case*, the word **realize** is used in this last sense; that is, it means to *accomplish* (or *achieve*) the use case.

An **interaction diagram** (**sequence diagram** or **communication diagram**) depicts the realization of a specific scenario of the use case. We first consider the use case Estimate Funds Available for Week.

**FIGURE 13.32**

The Estimate Funds Available for Week use case.

**MSG Staff Member**

* + 1. Estimate Funds Available for Week **Use Case** The use-case diagram of Figure 13.23 shows all the use cases. These include Estimate Funds Available for Week, which is shown separately in Figure 13.32. The

**MSG Foundation Information System**

Estimate Funds Available for Week

description of that use case was given in Figure 11.43, which is reproduced here as Figure 13.33 for convenience. From the description we deduce that, as reflected in the class diagram of Figure 13.34, the classes that enter into this use case are **User Interface Class**, which models the user interface; **Estimate Funds for Week Class**, the control class that models the computation of the estimate of the funds that are available to fund mortgages during that week; **Mortgage Class**, which models the estimated grants and payments for the week; **Investment Class**, which mod- els the estimated return on investments for the week; **MSG Application Class**, which models the estimated operating expenses for the week; and **Estimated Funds Report Class**, which models the printing of the report.

Figure 13.34 is a class diagram. That is, it shows the classes that participate in the realization of the use case and their relationships. A working software product, on the other hand, uses objects rather than classes. For example, a specific mortgage cannot be represented by **Mortgage Class** but rather by an object, a specific instance of **Mortgage Class**, denoted by **: Mortgage Class**. Also, the class diagram of Figure 13.34 shows the participating classes in the use case and their relationships; it does not show the sequence of events as they occur. Something more is needed to model a specific scenario such as the scenario of Figure 13.18, reproduced here as Figure 13.35.

Now consider Figure 13.36. This figure is a communication diagram (“collabora- tion diagram” in older versions of UML). It therefore shows the objects that interact as well as the messages that are sent, numbered in the order in which they are sent. A communication diagram depicts a realization of a specific scenario of a use case. In this case, Figure 13.36 depicts the scenario of Figure 13.35. In more detail, in the scenario the staff member wants to compute the funds available for the week. This is represented by message 1: Request estimate of funds available for week from **MSG Staff Member** to **: User Interface Class**, an instance of **User Interface Class**.

Next, this request is passed on to **: Estimate Funds for Week Class**, an instance of the control class that actually performs the calculation. This is repre- sented by message 2: Transfer request.

Four separate financial estimates are now determined by **: Estimate Funds for Week Class**. In step 1 of the scenario (Figure 13.35), the estimated annual return

**FIGURE 13.33** The description of the Estimate Funds Available for Week use case.

**Brief Description**

The Estimate Funds Available for Week use case enables an MSG Foundation staff member to estimate how much money the Foundation has available that week to fund mortgages.

**Step-by-Step Description**

* + - 1. For each investment, extract the estimated annual return on that investment. Summing the separate returns and dividing the result by 52 yields the estimated investment income for the week.
      2. Determine the estimated MSG Foundation operating expenses for the week by extracting the estimated annual MSG Foundation operating expenses and dividing by 52.
      3. For each mortgage:
         1. The amount to be paid this week is the total of the principal and interest payment and 1\_nd of the sum of the annual real-estate tax and the annual

52

homeowner’s insurance premium.

* + - * 1. Compute 28 percent of the couple’s current gross weekly income.
        2. If the result of Step 3.1 is greater than the result of Step 3.2, then the mortgage payment for this week is the result of Step 3.2, and the amount of the grant for this week is the difference between the result of Step 3.1 and the result of Step 3.2.
        3. Otherwise, the mortgage payment for this week is the result of Step 3.1, and there is no grant this week.
      1. Summing the mortgage payments of Steps 3.3 and 3.4 yields the estimated total mortgage payments for the week.
      2. Summing the grant payments of Step 3.3 yields the estimated total grant payments for the week.
      3. Add the results of Steps 1 and 4 and subtract the results of Steps 2 and 5. This is the total amount available for mortgages for the current week.
      4. Print the total amount available for new mortgages during the current week.

on investments is summed for each investment and the result divided by 52. This extraction of the estimated weekly return is modeled in Figure 13.36 by message 3: Request estimated return on investments for week from **: Estimate Funds for Week Class** to **: Investment Class** followed by message 4: Return esti- mated weekly return on investments in the reverse direction, that is, back to the object that is controlling the computation.

In step 2 of the scenario (Figure 13.35), the weekly operating expenses are esti- mated by taking the estimated annual operating expenses and dividing by 52. This extraction of the weekly return is modeled in Figure 13.36 by message 5: Request estimated operating expenses for week from **: Estimate Funds for Week Class** to **: MSG Application Class** followed by message 6: Return estimated operating expenses for week in the other direction.

In steps 3, 4, and 5 of the scenario (Figure 13.35), two estimates are determined, namely the estimated grants for the week and the estimated payments for the week. This is modeled in Figure 13.36 by message 7: Request estimated grants and

**FIGURE 13.34**

Class diagram showing the classes that realize the Estimate Funds Available for Week use case of the MSG Foundation case study.

**MSG Staff Member**

**Mortgage Class**

**User Interface Class**

**Estimate Funds for Week Class**

**MSG Application Class**

**Investment Class**

**Estimated Funds Report Class**



**FIGURE 13.35** A scenario of the Estimate Funds Available for Week use case.

An MSG Foundation staff member wishes to determine the funds available for mortgages this week.

1. For each investment, the information system extracts the estimated annual return on that investment. It sums the separate returns and divides the result by 52 to yield the estimated investment income for the week.
2. The information system then extracts the estimated annual MSG Foundation operating expenses and divides the result by 52.
3. For each mortgage:
   1. The information system computes the amount to be paid this week by adding the principal and interest payment to 1\_nd of the sum of the annual real-estate tax and the annual

52

homeowner’s insurance premium.

* 1. It then computes 28 percent of the couple’s current gross weekly income.
  2. If the result of Step 3.1 is greater than the result of Step 3.2, then it determines the mortgage payment for the week as the result of Step 3.2, and the amount of the grant for this week as the difference between the result of Step 3.1 and the result of Step 3.2.
  3. Otherwise, it takes the mortgage payment for this week as the result of Step 3.1, and there is no grant for the week.

1. The information system sums the mortgage payments of Steps 3.3 and 3.4 to yield the estimated total mortgage payments for the week.
2. It sums the grant payments of Step 3.3 to yield the estimated total grant payments for the week.
3. The information system adds the results of Steps 1 and 4 and subtracts the results of Steps 2 and 5. This is the total amount available for mortgages for the current week.
4. Finally, the software product prints the total amount available for new mortgages during the current week.

**FIGURE 13.36** A communication diagram of the realization of the scenario of Figure 13.35 of the Estimate Funds Available for Week use case of the MSG Application case study.

**: Mortgage Class**

8: Return estimated grants and payments for week

3: Request estimated return on investments for week

**: Investment**

1: Request estimate of funds available for week

7: Request estimated grants and payments for week

2: Transfer request

9: Compute estimated amount available for week

**Class**

4: Return estimated weekly return on investments

**MSG Staff**



15: Display suc- cessful completion

**: User**

14: Transfer suc-

cessful completion **: Estimate**

**Member**

message

**Interface**

**Class**

message

**Funds for Week Class**

6: Return estimated operating expenses for week

13: Send successful completion message

5: Request estimated operating expenses for week

10: Transfer estimated amount available for week

11: Print estimated amount available

**: MSG**

**Application Class**

12: Send

successful completion message

**: Estimated Funds Report Class**



payments for week from **: Estimate Funds for Week Class** to **: Mortgage Class** and by message 8: Return estimated grants and payments for week in the reverse direction.

Now the arithmetic computation of step 6 of the scenario is performed. This is mod- eled in Figure 13.36 by message 9: Compute estimated amount available for week. This is a self call, that is, **: Estimate Funds for Week Class** tells itself to perform the calculation. The result of the computation is stored in **: MSG Application Class** by message 10: Transfer estimated amount available for week.

Next, the result is printed in step 7 of the scenario (Figure 13.35). This is modeled in Figure 13.36 by message 11: Print estimated amount available from **: MSG Application Class** to **: Estimated Funds Report Class**.

Finally, an acknowledgment is sent to the MSG staff member that the task has been successfully completed. This is modeled in Figure 13.36 by messages 12: Send successful completion message, 13: Send successful completion message, 14: Transfer successful completion message, and 15: Display successful com- pletion message.

**FIGURE 13.37** The flow of events of the communication diagram of Figure 13.36 of the realization of the scenario of Figure 13.35 of the Estimate Funds Available for Week use case of the MSG Application case study.

An MSG staff member requests an estimate of the funds available for mortgages for the week (1, 2). The information system estimates the return on investments for the week (3, 4), the operating expenses for the week (5, 6), and the grants and payments for the week (7, 8). Then it estimates (9), stores (10), and prints out (11–15) the funds available for the week.

No client is going to approve the specification document unless he or she under- stands precisely what the proposed software product will do. For this reason, a written description of the communication diagram is essential. This is shown in Figure 13.37, the **flow of events**. Finally, the equivalent sequence diagram of the realization of the scenario is shown in Figure 13.38. When constructing a software product, either a communication diagram or a sequence diagram may prove to give better insight of a realization of a use case. In some situations, both are needed to get a full under- standing of a specific realization of a given use case. That is why, in this chapter, every communication diagram is followed by the equivalent sequence diagram. The sequence diagram of Figure 13.38 is fully equivalent to the communication diagram of Figure 13.36, so its flow of events is also shown in Figure 13.37.

The strength of a sequence diagram is that it shows the flow of messages unam- biguously. The order of the messages is particularly clear, as are the sender and receiver of each individual message. So, when the transfer of information is the focus of attention (which is the case for much of the time when performing the analy- sis workflow), a sequence diagram is superior to a communication diagram. On the other hand, the similarity between a sequence diagram (such as Figure 13.38) and the communication diagram that realizes the relevant scenario (such as Figure 13.36) is strong. Accordingly, on those occasions when the developers are concentrating on the classes, a communication diagram is generally more useful than the equivalent sequence diagram.

Summarizing, Figures 13.32 through 13.38 do not depict a random collection of UML artifacts. On the contrary, these figures depict a use case and artifacts derived from that use case. In more detail:

* Figure 13.32 depicts the use case Estimate Funds Available for Week. That is, Figure 13.32 models all possible sets of interactions, between the actor **MSG Staff Member** (an entity that is external to the software product) and the MSG Foundation software product itself, that relate to the action of estimating funds available for the week.
* Figure 13.33 is the description of that use case; that is, it provides a written account of the details of the Estimate Funds Available for Week use case of Figure 13.32.
* Figure 13.34 is a class diagram showing the classes that realize the Estimate Funds Available for Week use case. The class diagram depicts the classes that are needed to model all possible scenarios of the use case, together with their interactions.

**FIGURE 13.38** A sequence diagram of the realization of the scenario of Figure 13.35 of the Estimate Funds Available for Week use case of the MSG Application case study. This sequence diagram is fully equivalent to the communication diagram of Figure 13.36, so its flow of events is also shown in Figure 13.37.

**: User**

**: Estimate**

**: Investment**

**: MSG**

**: Mortgage**

**: Estimated**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **MSG Staff** | **Interface** | **Funds for** | **Class** | **Application** | **Class** | **Funds** |
| **Member** | **Class** | **Week Class** |  | **Class** |  | **Report Class** |

1: Request estimate of

funds available for week

2: Transfer request

3: Request estimated return on invest- ments for week

4: Return estimated return on invest- ments for week

5: Request estimated operating expenses for week

6: Return estimated operating expenses for week

7: Request estimated grants and payments for week

8: Return estimated grants and payments for week 9: Compute estimated amount

available for week

10: Transfer estimated amount 11: Print estimated amount available for week available

12: Send successful com- pletion message

13: Send successful completion

message

14: Transfer suc- cessful completion message

15: Display suc- cessful completion message

**FIGURE 13.39**

The Manage an Asset use case.

* Figure 13.35 is a scenario, that is, one specific instance of the use case of Figure 13.32.
* Figure 13.36 is a communication diagram of the realization of the scenario of Figure 13.35; that is, it depicts the objects and the messages sent between them in the realization of that one specific scenario.
* Figure 13.37 is the flow of events of the communication diagram of the realization of the scenario of Figure 13.35. That is, just as Figure 13.33 is a written description of the Estimate Funds Available for Week use case of Figure 13.32, Figure

13.37 is a written description of the realization of the scenario of Figure 13.35.

* Figure 13.38 is the sequence diagram that is fully equivalent to the communication diagram of Figure 13.36. That is, the sequence diagram depicts the objects and the messages sent between them in the realization of the scenario of Figure 13.35. Its flow of events is therefore also shown in Figure 13.37.

It has been stated many times in this book that the Unified Process is use-case driven. These bulleted items explicitly state the precise relationship between each of the artifacts of Figures 13.33 through 13.38 and the use case of Figure 13.32 that underlies each of them.

* + 1. Manage an Asset **Use Case**

The Manage an Asset use case is shown in Figure 13.39 and its description in Figure 13.40. A class diagram showing the classes that realize the Manage an Asset use case is shown in Figure 13.41. Initially it was assumed that only one control



**FIGURE 13.40**

**Step-by-Step Description**

1. Add, modify, or delete an investment or mortgage, or update the borrower’s weekly income.

**Brief Description**

The Manage an Asset use case enables an MSG Foundation staff member to add and delete assets and manage the portfolio of assets (investments and mortgages). Managing a mortgage includes updating the weekly income of a couple who have borrowed money from the Foundation.

Description of

the Manage an Asset use case.

**MSG Staff Member**

**Borrowers**

**MSG Foundation Information System**

Manage an Asset

**FIGURE 13.41**

A class diagram showing the classes that realize the Manage an Asset use case of the MSG Foundation case study.

**FIGURE 13.42**

An MSG Foundation staff member wants to update the annual real-estate tax on a home for which the Foundation has provided a mortgage.

1. The staff member enters the new value of the annual real- estate tax.
2. The information system updates the date on which the annual real-estate tax was last changed.

A scenario of the Manage an Asset use case.

**Borrowers**

**MSG Staff Member**

**User Interface Class**

**Mortgage Class**

In some scenarios, the borrowers tell the

MSG staff member their current weekly income.

**Manage an Asset Class**

**Investment Class**



class was needed (see Figure 13.31). However, Figure 13.41 shows that a second control class, **Manage an Asset Class**, is required; additional control classes may have to be added in subsequent iterations.

The normal part of the extended scenario of Figure 13.16 of the use case Manage a Mortgage (and hence of Manage an Asset) is reproduced as Figure 13.42. In this scenario, an MSG staff member updates the annual real- estate tax on a mortgaged home and the software product updates the date on which the tax was last changed. Figure 13.43 is the communication diagram of this scenario. Notice that object **: Investment Class** does not play an active role in this communication diagram because the scenario of Figure 13.42 does not

**FIGURE 13.43** A communication diagram of the realization of the scenario of Figure 13.42 of the Manage an Asset use case of the MSG Foundation case study.

**: Mortgage Class**

**Borrowers**

3: Update tax and date

1: Update annual

real-estate tax 2: Transfer data

4: Send successful completion message

**MSG Staff Member**

6: Display successful completion message

**: User Interface Class**

5: Send successful completion message

**: Manage an Asset Class**

**: Investment Class**



involve an investment, only a mortgage. Also, the **Borrowers** do not play a role in this scenario either. The flow of events is left as an exercise (Problem 13.14). The sequence diagram equivalent to the communication diagram of Figure 13.43 is shown in Figure 13.44.

Now consider a different scenario of the use case Manage an Asset (Figure 13.39), namely, the extended scenario of Figure 13.17, the normal part of which is repro- duced here as Figure 13.45. In this scenario, at the request of the borrowers, the MSG staff member updates the weekly income of a couple who have an MSG mort- gage. As explained in Section 11.7, the scenario is initiated by the **Borrowers**, and their data are entered into the software product by the **MSG Staff Member**, as stated in the note in the communication diagram of Figure 13.46. The flow of events is again left as an exercise (Problem 13.15). The equivalent sequence diagram is shown in Figure 13.47.

**FIGURE 13.44** A sequence diagram of the realization of the scenario of Figure 13.42 of the Manage an Asset use case of the MSG Foundation case study.

**Borrowers**

**MSG Staff**

**Member**

**: User Interface**

**Class**

**: Manage an**

**Asset Class**

**: Mortgage**

**Class**

**: Investment**

**Class**

1: Update annual real-estate tax

6: Display

successful com- pletion message

2: Transfer data

5: Send successful completion message

3: Update tax and date

4: Send successful completion message



**FIGURE 13.45** A second scenario of the Manage an Asset use case.

There is a change in the weekly income of a couple who have borrowed money from the MSG Foundation. They wish to have their weekly income updated

in the Foundation records by an MSG staff member so that their mortgage payments will be correctly computed.

1. The staff member enters the new value of the weekly income.
2. The information system updates the date on which the weekly income was last changed.

Comparing the interaction diagrams of Figures 13.43 and 13.46 (or, equivalently, the sequence diagrams of Figures 13.44 and 13.47), we see that, other than the actors involved, the only other difference between the two diagrams is that messages 1, 2, and 3 involve annual real-estate tax in the case of Figure 13.43 (or Figure 13.44) and weekly income in the case of Figure 13.46 (or Figure 13.47). This example highlights the difference between a use case, scenarios (instances of the use case), and communication or sequence diagrams of the realization of different scenarios of that use case.

Boundary class **User Interface Class** appears in all the realizations considered so far. In fact, the same screen will be used for all commands of the software product.

**FIGURE 13.46** A communication diagram of the realization of the scenario of Figure 13.45 of the Manage an Asset use case of the MSG Foundation case study.

**Borrowers**

The borrowers tell

the MSG staff member their current weekly income

**: Mortgage Class**

1: Update weekly income

3: Update

income and date

2: Transfer data

4: Send successful completion message

**MSG Staff**

6: Display successful completion **: User**

5: Send successful completion

**: Manage an Asset Class**

**Member**

message

**Interface**

**Class**

message

**: Investment Class**



An MSG staff member clicks on the appropriate operation in the revised menu of Figure 13.48. (The corresponding textual interface, as implemented in Appendices H and I, is given in Figure 13.49.)

* + 1. Update Estimated Annual Operating Expenses **Use Case**

The use case Update Estimated Annual Operating Expenses is shown in Figure 11.17 with a description in Figure 11.18. A class diagram showing the classes that realize the Update Estimated Annual Operating Expenses use case appears in Figure 13.50 and a communication diagram of a realization of a scenario of the use case in Figure 13.51. The equivalent sequence diagram is shown in Figure 13.52. Details of the scenario and the flow of events are left as an exercise (Problems 13.16 and 13.17).

**FIGURE 13.47** A sequence diagram of the realization of the scenario of Figure 13.45 of the Manage an Asset use case of the MSG Foundation case study.

**Borrowers**

1: Update weekly income

2: Transfer data

the MSG staff member their current weekly income

3: Update income and date

6: Display

successful com- pletion message

4: Send successful completion message

5: Send successful completion message

borrowers tell

The

**FIGURE 13.48**



*Quit*

Click on your choice:

*Estimate funds for the week Manage a mortgage Manage an investment*

*Update estimated annual operating expenses Produce a mortgages report*

*Produce an investments report*

Revised menu of the target MSG

Foundation case study.

**FIGURE 13.49**

MAIN MENU

MARTHA STOCKTON GREENGAGE FOUNDATION

1. Estimate funds available for week
2. Manage a mortgage
3. Manage an investment
4. Update estimated annual operating expenses
5. Produce a mortgages report
6. Produce an investments report
7. Quit

Type your choice and press <ENTER>:

Textual version

of the revised 13.48.

menu of Figure

**MSG Staff**

**Member**

**: User Interface**

**Class**

**: Manage an**

**Asset Class**

**: Mortgage**

**Class**

**: Investment**

**Class**

**FIGURE 13.50** A class diagram showing the classes that realize the Update Estimated Annual Operating Expenses use case of the MSG Foundation case study.

**MSG Staff Member**

**User Interface Class**

**MSG Application Class**

**FIGURE 13.51** A communication diagram of the realization of a scenario of the Update Estimated Annual Operating Expenses use case of the MSG Foundation case study.

1: Update annual expenses

4: Display successful

2: Update expenses and date

3: Send successful

**MSG Staff Member**

completion message

**: User Interface Class**

completion message

**: MSG Application Class**

**FIGURE 13.52** A sequence diagram of the realization of a scenario of the Update Estimated Annual Operating Expenses use case of the MSG Foundation case study.

**MSG Staff Member**

**: User Interface**

**Class**

**: MSG Application**

**Class**



1: Update annual expenses

2: Update expenses and date

4: Display

successful

completion message

3: Send

successful

completion message

**FIGURE 13.53**

The Produce a Report use case.

* + 1. Produce a Report **Use Case**

Use case Produce a Report is shown in Figure 13.53. The description of use case Produce a Report of Figure 11.39 is reproduced here as Figure 13.54. A class diagram showing the classes that realize the Produce a Report use case is shown in Figure 13.55.

**MSG Foundation Information System**

Produce a Report

**MSG Staff Member**



**FIGURE 13.54** Description of the Produce a Report use case.

**Brief Description**

The Produce a Report use case enables an MSG Foundation staff member to print a listing of all investments or all mortgages.

**Step-by-Step Description**

* + - 1. The following reports must be generated:
         1. Investments report—printed on demand:

The information system prints a list of all investments. For each investment, the following attributes are printed:

Item number Item name

Estimated annual return

Date estimated annual return was last updated

* + - * 1. Mortgages report—printed on demand:

The information system prints a list of all mortgages. For each mortgage, the following attributes are printed:

Account number Name of mortgagees Original price of home

Date mortgage was issued Principal and interest payment

Current combined gross weekly income

Date current combined gross weekly income was last updated Annual real-estate tax

Date annual real-estate tax was last updated Annual homeowner’s insurance premium

Date annual homeowner’s insurance premium was last updated

**FIGURE 13.55** A class diagram showing the classes that realize the Produce a Report use case of the MSG Foundation case study.

**MSG Staff Member**

**Mortgage Class**

**User Interface Class**

**Investment Class**

**Mortgages Report Class**

**Investments Report Class**

First consider the scenario of Figure 13.19 for listing all mortgages, reproduced here as Figure 13.56. A communication diagram of the realization of this scenario is shown in Figure 13.57. This realization models the listing of all mortgages. Accordingly, object **: Investment Class**, an instance of the other subclass of **Asset Class**, plays no role in this realization, and neither does **: Investments Report Class**. The flow of events is left as an exercise (Problem 13.18). The equiv- alent sequence diagram is shown in Figure 13.58.

Now consider the scenario of Figure 13.20 for listing all investments, reproduced here as Figure 13.59. A communication diagram of the realization of this scenario is shown in Figure 13.60. As opposed to the previous realization, Figure 13.60 models

**FIGURE 13.56** A scenario of the Produce a Report

use case.

An MSG staff member wishes to print a list of all mortgages.

1. The staff member requests a report listing all mortgages.

**FIGURE 13.57** A communication diagram of the realization of the scenario of Figure 13.56 of the Produce a Report use case of the MSG Foundation case study.

3: Print list of mortgages

**: Mortgage Class**

4: Send successful completion message

**: Mortgages Report Class**

1: Request list of mortgages

2: Transfer request

5: Send successful completion message

**MSG Staff Member**

6: Display successful completion message

**: User Interface Class**

**: Investment Class**

**: Investments Report Class**

**FIGURE 13.58** A sequence diagram of the realization of the scenario of Figure 13.56 of the Produce a Report use case of the MSG Foundation case study.

**MSG Staff Member**

**: User Interface Class**

**: Mortgage Class**

**: Mortgages**

**Report Class**

**: Investment Class**

**: Investments**

**Report Class**

1: Request list of mortgages

2: Transfer request

6: Display

successful

completion message

5: Send successful

completion message

3: Print list of mortgages

4: Send successful

completion message



**FIGURE 13.59** Another scenario of the Produce a Report use case.

An MSG staff member wishes to print a list of all investments.

1. The staff member requests a report listing all investments.

the listing of the investments; mortgages are ignored here. The equivalent sequence diagram is shown in Figure 13.61.

This concludes the realization of the four use cases of Figure 13.23, the eighth iteration of the use-case diagram of the MSG Foundation case study.

**FIGURE 13.60**

A communication diagram of the realization of the scenario of Figure 13.59 of the Produce a Report use case of the MSG Foundation case study.

**MSG Staff Member**

1: Request list

of investments

6: Display successful completion message

**: Mortgage Class**

**: User Interface Class**

**: Mortgages Report Class**

5: Send successful completion message

2: Transfer request

3: Print list of investments

**: Investment Class**

4: Send successful completion message

**: Investments Report Class**

**FIGURE 13.61** A sequence diagram of the realization of the scenario of Figure 13.59 of the Produce a Report use case of the MSG Foundation case study.

**MSG Staff Member**

1: Request list

of investments

6: Display

successful

completion message

**: User Interface Class**

**: Mortgage**

**Class**

**: Mortgages**

**Report Class**

**: Investment Class**

**: Investments**

**Report Class**



|  |  |  |
| --- | --- | --- |
| 2: Transfer request |  |  |
|  |  | 5: Send successful |
|  |  | completion message |

**453**

3: Print list of investments

4: Send successful

completion message

###### ase Study

*C*

13.16

**FIGURE 13.62**

Class diagram combining the class diagrams of 13.34, 13.41,

13.50, and

13.55.

Incrementing the Class Diagram:

The MSG Foundation Case Study

The entity classes were extracted in Sections 13.9 through 13.12, yielding Figure 13.29, which shows four entity classes. The boundary classes were extracted in Section 13.13 and the control classes in Sections 13.14 and 13.15.2. In the course of realizing the vari- ous use cases in Section 13.15, interrelationships between many of the classes became apparent; these interrelationships are reflected in the class diagrams of Figures 13.34, 13.41, 13.50, and 13.55. Figure 13.62 combines these class diagrams.

Now the class diagrams of Figures 13.29 and 13.62 are combined to yield the fourth iteration of the class diagram of the MSG Foundation case study, shown in

**MSG Staff Member**

**User Interface Class**

**Manage an Asset Class**

**Estimate Funds for Week Class**

**Investment Class**

**Mortgage Class**

**MSG Application Class**

**Investments Report Class**



**Mortgages Report Class**

**Estimate Funds Report Class**

**FIGURE 13.63**

The fourth iteration of the class diagram of the MSG Foundation case study, obtained by combining the class diagrams

of Figures 13.29

and 13.62.

**MSG Staff Member**

**Investment Class**

**User Interface Class**

**Manage an Asset Class**

**Estimate Funds for Week Class**

**Investments Report Class**

**Asset Class**

**Mortgage Class**

**Mortgages Report Class**

**MSG Application Class**

**Estimated Funds Report Class**



Relationships in Figure 13.62

Relationships in Figure 13.29

Figure 13.63. More specifically, starting with Figure 13.62, **Asset Class** of Figure 13.29 is added. Then the two inheritance (generalization) relationships in Figure 13.29 are drawn in; they are shown with dashed lines to distinguish them. The result, Figure 13.63, the fourth iteration of the class diagram, is the class diagram at the end of the analysis workflow.

The last step of the analysis workflow of the MSG Foundation case study is to draw up the software project management plan (this is done during the elaboration phase; see Section 3.10.2). Appendix F contains a software project management plan for the development of the MSG Foundation product by a small (three-person) soft- ware organization.

###### ase Study



*C*

13.17

The Test Workflow:

The MSG Foundation Case Study

The analysis workflow of the MSG Foundation case study is checked in two ways. First the entity classes are checked using CRC cards, as described in Section 13.7. Then all the artifacts of the analysis workflow are inspected (Section 6.2.3).

This concludes the analysis workflow of the MSG Foundation case study.

* 1. **The Specification Document in the Unified Process**

A primary goal of the analysis workflow is to produce the **specification document**, but at the end of Section 13.17 it was claimed that the analysis workflow is now complete. The obvious question is, Where is the specification document?

The short answer is, the Unified Process is use-case driven. In more detail, the use cases and the artifacts derived from them contain all the information that, in the traditional paradigm, appears in the specification document in text form, and more.

For example, consider the use case Estimate Funds Available for Week. When the requirements workflow is performed, the Estimate Funds Available for Week use case (Figure 11.27) and its description (Figure 11.40) are shown to the client, the trustees of the MSG Foundation. The developers must be meticulous in ensuring that the trustees fully understand these two artifacts and agree that these artifacts accurately model the software product the Foundation needs. Then, during the analysis workflow, the trustees are shown the use case Estimate Funds Available for Week (Figure 13.32), its description (Figure 13.33), the class diagram showing the classes that realize the use case (Figure 13.34), a scenario of the use case (Figure 13.35), the interaction diagrams of the real- ization of a scenario of the use case (Figures 13.36 and 13.38), and the flow of events of these interaction diagrams (Figure 13.37).

The set of artifacts just listed all appertain to only the use case Estimate Funds Available for Week. As shown in Figure 13.23, there are four use cases altogether. The same set of artifacts are produced for each of the scenarios of each of the use cases. The resulting collection of artifacts, some diagrammatic and some textual, convey to the client more information more accurately than the purely textual specification document of the traditional paradigm possibly could.

The traditional specification document usually plays a contractual role. That is, once it has been signed by both the developers and the client, it essentially constitutes a legal document. If the developers build a software product that satisfies the specification docu- ment, the client is obligated to pay for the software product, and conversely, if the product does not conform to its specification document, the developers are required to fix it if they want to get paid. In the case of the Unified Process, the collection of artifacts of all

the scenarios of all the use cases similarly constitutes a contract. Therefore, as claimed at the end of Section 13.17, the analysis workflow of the MSG Foundation case study is indeed complete.

As stated before, the Unified Process is use-case driven. When using the Unified Pro- cess, instead of constructing a rapid prototype, the use cases, or more precisely, interaction diagrams reflecting the classes that realize the scenarios of the use cases, are shown to the client. The client can understand how the target software product will behave just as well from the interaction diagrams and their written flow of events as from a rapid prototype. After all, a scenario is a particular execution sequence of the proposed software product, as is each execution of the rapid prototype. The difference is that the rapid prototype is gener- ally discarded, whereas the use cases are successively refined, with more information added each time.

However, there is one area where a rapid prototype is superior to a scenario, the user interface. This does not mean that a rapid prototype should be built just so that specimen screens and reports can be examined by the client and users. But specimen screens and reports need to be constructed, as described in Section 11.13, preferably with the aid of CASE tools such as screen generators and report generators (Section 5.5).

In Section 13.19, methods for determining actors and use cases are provided.

* 1. More on Actors and Use Cases

As stated in Section 11.4.3, a use case depicts an interaction between the software product itself and the actors (the users of that software product). Now that a number of examples of actors and use cases have been presented, it is appropriate to describe how to find actors and use cases.

To find the actors, we have to consider every **role** in which an individual can interact with the software product. For example, consider a couple who wish to obtain a mortgage from the MSG Foundation. When they apply for the mortgage, they are **Applicants**, whereas after their application has been approved and money to buy their home loaned to them, they become **Borrowers**. In other words, actors are not so much individuals as roles played by those individuals. In our example, the actors are not the couple, but rather first the couple playing the role of **Applicants** and then the couple playing the role of **Borrowers**. This means that merely listing all the individuals who will use the software product is not a satisfactory way of finding the actors. Instead, we need to find all the roles played by each user (or group of users). From the list of roles we can extract the actors.

In the terminology of the Unified Process, the term **worker** is used to denote a par-

ticular role played by an individual. This is a somewhat unfortunate term, because the word *worker* usually refers to an employee. In the terminology of the Unified Process, in the case of a couple with a mortgage, **Applicants** and **Borrowers** are two dif- ferent workers. In this book, in the interests of clarity the word *role* is used in place of *worker*.

Within a business context, the task of finding the roles is generally straightforward. The use-case business model usually displays all the roles played by the individuals who inter- act with the business, thereby highlighting the business actors. We then find the subset of



* **Iterate**

Perform functional modeling. Perform entity class modeling. Perform dynamic modeling.

* **Until** the entity classes have been satisfactorily extracted.
* Extract the boundary classes and control classes.
* Refine the use cases.
* Perform use-case realization.

**Box 13.1**

**How to Perform Object-Oriented Analysis**

the use-case business model that corresponds to the use-case model of the requirements. In more detail,

* + 1. Construct the use-case business model by finding all the roles played by the individuals who interact with the business.
    2. Find the subset of the use-case diagram of the business model that models the software product we wish to develop. That is, consider only those parts of the business model that correspond to the proposed software product.

Once the actors have been determined, finding the use cases is generally straightfor- ward. For each role, there are one or more use cases. So, the starting point in finding the use cases of the requirements is finding the actors, as described in this section.

How to Perform Box 13.1 summarizes object-oriented analysis.

* 1. **CASE Tools for the Object-Oriented**

**Analysis Workflow**

Bearing in mind the role played by diagrams in object-oriented analysis, it is not surprising that a number of CASE tools have been developed to support object-oriented analysis. In its basic form, such a tool is essentially a drawing tool that makes it easy to perform each of the modeling steps. More important, it is far simpler to modify a diagram constructed with a drawing tool than to attempt to change a hand-drawn figure. Accordingly, a CASE tool of this type supports the graphical aspects of object-oriented analysis. In addition, some tools of this type not only draw all the relevant diagrams but CRC cards as well. A strength of these tools is that a change to the underlying model is reflected automatically in all the affected diagrams; after all, the various diagrams are merely different views of the underlying model.

On the other hand, some CASE tools support not just object-oriented analysis but a con- siderable portion of the rest of the object-oriented life cycle as well. Nowadays virtually all of these tools support UML [Rumbaugh, Jacobson, and Booch, 1999]. Examples of such

tools include IBM Rational Rose and Together. ArgoUML is a typical open-source CASE tool of this type.

* 1. Metrics for the Object-Oriented Analysis Workflow

As with the other core workflows, during object-oriented analysis it is essential to measure the five fundamental metrics: size, cost, duration, effort, and quality. One measure of the size of the object-oriented analysis is the number of pages of UML diagrams; this metric can be used to compare different projects.

With regard to quality, as with classical analysis, it is essential to keep accurate fault statistics. Also, the rate at which faults are detected can give a measure of the efficiency of the inspection process.

* 1. Challenges of the Object-Oriented Analysis Workflow

Object-oriented analysis is a specific approach to analysis, so the challenges of classical analysis described in Section 12.16 apply equally to object-oriented analysis. In particu- lar, the second challenge listed in that section is that it is easy to cross the boundary line between specifications (what) and design (how). This danger is especially acute in the case of object-oriented analysis.

Recall that, as described in Section 1.9, the transition from object-oriented analysis to object-oriented design is far smoother than the transition in the classical paradigm from the analysis phase to the design phase. In the classical paradigm, an initial task of the design phase is to decompose the product into modules. In contrast, the classes, the “modules” of the object-oriented design workflow, are extracted during the object-oriented analysis workflow, ready for refinement during the object-oriented design workflow. The presence of classes from early in the OOA workflow means that the temptation to carry the OOA too far can be extremely strong.

For example, consider the issue of allocation of methods to classes. One task of the clas- sical analysis phase is to determine the data and operations of the target product. However, allocation of the various operations to specific modules should be delayed until the classi- cal design phase, because as pointed out in Section 12.16, we first have to determine how the product as a whole is broken down into modules.

In the object-oriented paradigm, however, this latter task is part of the analysis workflow. That is, during the object-oriented analysis workflow, we determine the modules (classes) and their interactions; the result is depicted in the class diagram. Therefore, there is no apparent reason why we should wait until the object-oriented design workflow before allo- cating methods to classes.

Nevertheless, it is important to remember that object-oriented analysis is an iterative process. In the course of refining the various models, frequently large portions of the class diagram have to be reorganized. Reallocating the methods then results in unnecessary additional rework.

At each step of the OOA process it is a good idea to minimize the information that would have to be reorganized during iteration. Therefore, allocation of methods to classes should wait until the design workflow, no matter how tempting it may be to go just a little further during the object-oriented analysis workflow.

**Chapter Review**

Object-oriented analysis is introduced (Section 13.1). Extracting entity classes is described in Sec- tion 13.2. The technique is then applied to the elevator problem case study (Section 13.3); functional modeling, entity class modeling, and dynamic modeling are performed in Sections 13.4, 13.5, and 13.6, respectively. Next, object-oriented analysis aspects of the test workflow are covered in Section

13.7. Extraction of boundary and control classes is the subject of Section 13.8. The class extrac- tion of the MSG Foundation case study is described in Section 13.9 (the initial functional model), Section 13.10 (the initial class diagram), Section 13.11 (the initial dynamic model), Section 13.12 (revision of the entity classes), Section 13.13 (extraction of the boundary classes), and Section 13.14 (extraction of the control classes). Application of the Unified Process to the MSG Foundation case study resumes in Section 13.15 (realization of the use cases), Section 13.16 (class diagram incre- mentation), and Section 13.17 (test workflow). The specification document for the Unified Process is discussed in Section 13.18. Additional information regarding actors and use cases appears in Sec- tion 13.19. CASE tools and metrics for object-oriented analysis are described in Sections 13.20 and 13.21, respectively. The chapter concludes with a discussion of the challenges of the object-oriented analysis workflow (Section 13.22).

An overview of the MSG Foundation case study for Chapter 13 appears in Figure 13.64, and for the elevator problem in Figure 13.65.

**FIGURE 13.64** Overview of the MSG Foundation case study for Chapter 13.

|  |  |
| --- | --- |
| Initial functional model | Section 13.9 |
| Seventh iteration of the use-case diagram | Figure 13.15 |
| Initial class diagram | Section 13.10 |
| First iteration of the class diagram | Figure 13.21 |
| Second iteration of the class diagram | Figure 13.22 |
| Eighth iteration of the use-case diagram | Figure 13.23 |
| Second iteration of the class diagram, with attributes added | Figure 13.24 |
| Initial dynamic model | Section 13.11 |
| Initial statechart | Figure 13.25 |
| Revising the entity classes | Section 13.12 |
| Third iteration of the class diagram | Figure 13.28 |
| Extracting the boundary classes | Section 13.13 |
| Extracting the control classes | Section 13.14 |
| Use-case realization | Section 13.15 |
| Estimate Funds Available for Week use case | Section 13.15.1 |
| Manage an Asset use case | Section 13.15.2 |
| Update Estimated Annual Operating Expenses use case | Section 13.15.3 |
| Produce a Report use case | Section 13.15.4 |
| Incrementing the class diagram | Section 13.16 |
| Fourth iteration of the class diagram | Figure 13.63 |

**FIGURE 13.65** Overview of the elevator problem case study for Chapter 13.

|  |  |
| --- | --- |
| Object-oriented analysis | Section 13.3 |
| Functional modeling | Section 13.4 |
| Entity class modeling | Section 13.5 |
| First iteration of the class diagram | Figure 13.5 |
| Second iteration of the class diagram | Figure 13.6 |
| Dynamic modeling | Section 13.6 |
| First iteration of the statechart for the elevator controller | Figure 13.7 |
| Test workflow | Section 13.7 |
| Third iteration of the class diagram | Figure 13.10 |
| Fourth iteration of the class diagram | Figure 13.12 |
| First iteration of the statechart for the elevator subcontroller | Figure 13.13 |

**For Further Reading**

Fusion [Coleman et al., 1994] is a second-generation OOA technique, a combination (or fusion) of a number of first-generation techniques, including OMT [Rumbaugh et al., 1991] and Objectory [Jacob- son, Christerson, Jonsson, and Overgaard, 1992]. The Unified Software Development Process unifies the work of Jacobson, Booch, and Rumbaugh [1999]. Catalysis is another important object-oriented methodology [D’Souza and Wills, 1999].

ROOM is an object-oriented methodology for real-time software [Selic, Gullekson, and Ward, 1995]. Further information on real-time object-oriented technologies can be found in [Awad, Kuu- sela, and Ziegler, 1996].

Full details regarding UML can be found in [Booch, Rumbaugh, and Jacobson, 1999] and [Rum- baugh, Jacobson, and Booch, 1999]. The October 1999 issue of *Communications of the ACM* contains a broad variety of papers on the use of UML. UML is now under the control of the Object Manage- ment Group; the latest version of UML will be found at the OMG Website, [www.omg.org](http://www.omg.org/).

The noun-extraction technique used in this chapter to extract candidate classes is formalized in [Juristo, Moreno, and López, 2000]. CRC cards were first put forward in [Beck and Cunning- ham, 1989]. [Wirfs-Brock, Wilkerson, and Wiener, 1990] is a good source of information on CRC cards.

A number of comparisons of object-oriented analysis techniques have been published, including [de Champeaux and Faure, 1992], [Monarchi and Puhr, 1992], and [Embley, Jackson, and Woodfield, 1995]. A comparison of both object-oriented and classical analysis techniques appears in [Fichman and Kemerer, 1992].

Management of iteration in object-oriented projects is described in [Williams, 1996]. Statecharts are described in [Harel and Gery, 1997]. The reuse of specifications in the object-oriented paradigm is described in [Bellinzona, Fugini, and Pernici, 1995].

A variety of papers on formal techniques for object-oriented software appear in the July 2000 issue of *IEEE Transactions on Software Engineering*.

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

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

**Key Terms** abstract noun *411*

actor *407*

analysis workflow *405*

attribute *411*

backtrack *430*

boundary class *405*

class diagram *411*

class–responsibility– collaboration (CRC) cards *413*

communication diagram *435*

control class *406*

dynamic modeling *406*

entity class *405*

entity class modeling *406*

event *431*

exception scenario *408* flow of events *440* functional modeling *406*

interaction diagram *435*

legacy system *405*

millennium bug *405*

normal scenario *408*

noun-extraction method *411*

object-oriented analysis (OOA) *404*

realize (in the Unified Theory context) *435*

responsibility-driven design *408*

role *457*

scenario *406*

sequence diagram *435*

specification document *456*

state *418*

state variable *418*

statechart *414*

stereotype *406*

test workflow *417*

transition *431*

use case *407*

use-case realization *435*

worker *457*

Y2K problem *405*

**Problems1**

* 1. Modify the scenario of Figure 13.11 to reflect the fourth iteration of the class diagram of the elevator problem case study (Figure 13.12).
  2. Develop a statechart for the **Button Class** shown in Figure 13.12.
  3. Develop a statechart for the **Elevator Class** shown in Figure 13.12.
  4. Develop a statechart for the **Elevator Doors Class** shown in Figure 13.12.
  5. Construct a CRC card for the **Floor Subcontroller Class** shown in Figure 13.12.
  6. Why must the finite state machine formalism of Section 12.7 be changed when used for object- oriented analysis?
  7. What is the latest point in the analysis workflow in which classes can be introduced without adversely affecting the project?
  8. What is the earliest point in the Unified Process in which classes can meaningfully be intro- duced?
  9. Is it possible to represent the dynamic model using a formalism other than the statechart described in this chapter? Explain your answer.
  10. Why are the attributes of the classes but not the methods determined during object-oriented analysis?
  11. A noun-extraction process is described in Section 13.5.1. Why do we not also extract the verbs? And what about the other six parts of speech (adjectives, adverbs, conjunctions, inter- jections, prepositions, and pronouns)?
  12. Give an extended scenario of the use case Manage an Investment of Figures 11.30 and 11.31.
  13. Give an extended scenario of the use case Update Estimated Annual Operating Expenses of Figures 11.17 and 11.18.
  14. Give the flow of events of the interaction diagrams of Figures 13.43 and 13.44.
  15. Give the flow of events of the interaction diagrams of Figures 13.46 and 13.47.
  16. Check that your answer to Problem 13.13 is a possible scenario for the interaction diagrams of Figures 13.51 and 13.52. If not, modify your scenario.

1Problem 12.16 (Term Project) and Problems 12.20 and 12.21 (Case Study) can be done at the end of either Chapter 12 or Chapter 13.

Chapter 13 *Object-Oriented Analysis* **463**

* 1. Give the flow of events of the interaction diagrams of Figures 13.51 and 13.52.
  2. Give the flow of events of the interaction diagrams of Figures 13.57 and 13.58.
  3. (Analysis and Design Project) Perform the analysis workflow of the library software product of Problem 8.7.
  4. (Analysis and Design Project) Perform the analysis workflow of the product for determining whether a bank statement is correct of Problem 8.8.
  5. (Analysis and Design Project) Perform the analysis workflow of the automated teller machine of Problem 8.9. There is no need to consider the details of the constituent hardware compo- nents such as the card reader, printer, and cash dispenser. Instead, simply assume that, when the ATM sends commands to those components, they are correctly executed.
  6. (Term Project) Perform the analysis workflow of the Chocoholics Anonymous product described in Appendix A.
  7. (Case Study) Add **Report Class** to the analysis workflow of the MSG Foundation case study (Sections 13.9 through 13.16). Is this an improvement or an unnecessary complication?
  8. (Case Study) Determine what happens when object-oriented analysis starts with dynamic modeling. Start with the statechart of Figure 13.25 and complete the object-oriented analysis process for the MSG Foundation case study.
  9. (Case Study) Compare and contrast the structured systems analysis of the MSG Foundation case study of Section 12.4 with the object-oriented analysis workflow of Sections 13.9 through 13.11.
  10. (Readings in Software Engineering) Your instructor will distribute copies of [Juristo, Moreno, and López, 2000]. What is your opinion of their approach to object-oriented analysis?

**References**

[Awad, Kuusela, and Ziegler, 1996] M. AWAD, J. KUUSELA, AND J. ZIEGLER, *Object-Oriented Technol- ogy for Real-Time Systems: A Practical Approach Using OMT and Fusion,* Prentice Hall, Upper Saddle River, NJ, 1996.

[Beck and Cunningham, 1989] K. BECK AND W. CUNNINGHAM, “A Laboratory for Teaching Object- Oriented Thinking,” Proceedings of OOPSLA ’89, *ACM SIGPLAN Notices* **24** (October 1989), pp. 1–6.

[Bellinzona, Fugini, and Pernici, 1995] R. BELLINZONA, M. G. FUGINI, AND B. PERNICI, “Reusing Specifications in OO Applications,” *IEEE Software* **12** (March 1995), pp. 656–75.

[Booch, Rumbaugh, and Jacobson, 1999] G. BOOCH, J. RUMBAUGH, AND I. JACOBSON, *The UML Users Guide*, Addison-Wesley, Reading, MA, 1999.

[Coleman et al., 1994] D. COLEMAN, P. ARNOLD, S. BODOFF, C. DOLLIN, H. GILCHRIST, F. HAYES,

AND P. JEREMAES, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[D’Souza and Wills, 1999] D. D’SOUZA AND H. WILLS, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, Reading, MA, 1999.

[de Champeaux and Faure, 1992] D. DE CHAMPEAUX AND P. FAURE, “A Comparative Study of Object- Oriented Analysis Methods,” *Journal of Object-Oriented Programming* **5** (March–April 1992), pp. 21–33.

[Embley, Jackson, and Woodfield, 1995] D. W. EMBLEY, R. B. JACKSON, AND S. N. WOODFIELD, “OO

Systems Analysis: Is It or Isn’t It?” *IEEE Software* **12** (July 1995), pp. 18–33.

[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.

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

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

[Harel and Gery, 1997] D. HAREL AND E. GERY, “Executable Object Modeling with Statecharts,”

*IEEE Computer* **30** (July 1997), pp. 31–42.

[Jacobson, Booch, and Rumbaugh, 1999], I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jacobson, Christerson, Jonsson, and Overgaard, 1992] I. JACOBSON, M. CHRISTERSON, P. JONSSON, AND G. OVERGAARD, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, New York, 1992.

[Juristo, Moreno, and López, 2000] N. JURISTO, A. M. MORENO, AND M. LÓPEZ, “How to Use Linguis- tic Instruments for Object-Oriented Analysis,” *IEEE Software* **17** (May–June 2000), pp. 80–89.

[Monarchi and Puhr, 1992] D. E. MONARCHI AND G. I. PUHR, “A Research Typology for Object- Oriented Analysis and Design,” *Communications of the ACM* **35** (September 1992), pp. 35–47. [Rumbaugh et al., 1991] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN,

*Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rumbaugh, Jacobson, and Booch, 1999] J. RUMBAUGH, I. JACOBSON, AND G. BOOCH, *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading, MA, 1999.

[Selic, Gullekson, and Ward, 1995] B. SELIC, G. GULLEKSON, AND P. T. WARD, *Real-Time Object- Oriented Modeling*, John Wiley and Sons, New York, 1995.

[USNO, 2000] “The 21st Century and the Third Millennium—When Will They Begin?” U.S. Naval Observatory, Astronomical Applications Department, at aa.usno.navy.mil/AA/faq/docs/ millennium.html, February 22, 2000.

[Williams, 1996] J. D. WILLIAMS, “Managing Iteration in OO Projects,” *IEEE Computer* **29** (Septem- ber 1996), pp. 39–43.

[Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER,

*Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

Chapter 14