27

Object-oriented analysis

Focused initially on the implementation aspects of software construction, the object-oriented method quickly expanded to cover the totality of the software lifecycle. Of particular interest has been the application of O-O ideas to the modeling of software systems, or even of non-software systems and issues. This use of object technology to present **problems** rather than solutions is known as object-oriented analysis.

In the past few years, many books have appeared on the topic and many specific methods of object-oriented analysis have been proposed. The bibliography section lists some of the best-known books, and Web addresses for some of the best-known methods.

Most of the concepts introduced in the preceding chapters are directly relevant to object-oriented analysis. Here we will briefly review what make object-oriented *analysis* special among other object-oriented topics, and what makes *object-oriented* analysis different from other analysis methods.

Two points of terminology to avoid imagining differences where none exist. First, you will encounter, as a synonym for "analysis", the term *system modeling*, or just *modeling*. Second, the computing science community tends to use the word *specification* where information modeling folks talk about analysis; in particular, computing scientists have devoted considerable efforts to devising methods and languages for *formal specification* using mathematical techniques for purposes of system modeling. The goals are the same, although the techniques may differ. In the past few years the two communities — information modelers and formal specifiers — have been paying more attention to each other's contributions.

27.1 THE GOALS OF ANALYSIS

To understand analysis issues we must be aware of the roles of analysis in software development and define requirements on an analysis method.

Tasks

By devoting time to analysis and producing analysis documents we pursue seven goals:

Goals of performing analysis

A1 • To understand the problem or problems that the eventual software system, if any, should solve.

A2 • To prompt relevant questions about the problem and the system.

A3 • To provide a basis for answering questions about specific properties of the problem and system.

A4 • To decide what the system should do.

A5 • To decide what the system should not do.

A6 • To ascertain that the system will satisfy the needs of its users, and define acceptance criteria (especially when the system is developed for an outside customer under a contractual relationship).

A7 • To provide a basis for the development of the system.

If analysis is being applied to a non-software system, or independently of a decision to build a software system, A1, A2 and A3 may be the only relevant goals.

For a software system, the list assumes that analysis follows a stage of *feasibility study* which has resulted in a decision to build a system. If, as sometimes happens, the two stages are merged into one (not an absurd proposition, since you may need an in-depth analysis to determine whether a satisfactory result is conceivable), the list needs another item: A0, deciding whether to build a system.

Although related, the goals listed are distinct, prompting us in the rest of this chapter to look for a set of complementary techniques; what is good for one of the goals may be irrelevant to another.

Goals A2 and A3 are the least well covered in the analysis literature and deserve all the emphasis they can get. One of the primary benefits of an analysis process, independently of any document that it produces in the end, is that it leads you to ask the relevant questions (A2): what is the maximum acceptable temperature? What are the recognized categories of employees? How are bonds handled differently from stocks? By providing you with a framework, which you will have to fill using input from people competent in the application domain, an analysis method will help spot and remove obscurities and ambiguities which can be fatal to a development. Nothing is worse than discovering, at the last stage of implementation, that the marketing and engineering departments of the client company have irreconcilable views of what equipment maintenance means, that one of these views was taken by default, and that no one cared to check what the actual order giver had in mind. As to A3, a good analysis document will be the place to which everyone constantly goes back if delicate questions or conflicting interpretations arise during the development process.

Requirements

The practical requirements on the analysis process and supporting notations follow from the above list of goals:

- There must be a way to let non-software people contribute input to the analysis, examine the results and discuss them (A1, A2).
- The analysis must also have a form that is directly usable by software developers (A7).
- The approach must *scale up* (A1).
- The analysis notation must be able to express precise properties unambiguously (A3).
- It must enable readers to get a quick glimpse of the overall organization of a system or subsystem (A1, A7)

Scaling up (the third point) means catering to systems that are complex, large or both — the ones for which you most need analysis. The method should enable you to describe the high-level structure of the problem or system, and to organize the description over several layers of abstraction, so that you can at any time focus on as big or as small a part of the system as you wish, while retaining the overall picture. Here, of course, the structuring and abstracting facilities of object technology will be precious.

Scaling up also means that the criteria of extendibility and reusability, which have guided much of our earlier discussions, are just as applicable to analysis as they are to software design and implementation. Systems change, requiring their descriptions to follow; and systems are similar to previous systems, prompting us to use *libraries* of specification elements to build their specifications, just as we use libraries of software components to build their implementations.

The clouds and the precipice

It is not easy to reconcile the last two requirements of the above list. The conflict, already discussed in the context of abstract data types, has plagued analysis methods and specification languages as long as they have existed. How do you "express precise properties unambiguously" without saying too much? How do you provide readable broad-brush structural descriptions without risking vagueness?

The analyst walks on a mountain path. On your left is the mountain top, deep ensconced in clouds; this is the realm of the fuzzy. But you must also stay away, on your right, from the precipice of overspecification, to which you might be perilously drawn if your attempts to be precise tempt you to say too much, especially by giving out *implementation* details instead of external properties of the system.

The risk of overspecification is ever present in the minds of people interested in analysis. (It is said that, to gain the upper hand in a debate in this field, you should try "*Approach X is nice, but isn't it a tad implementation-oriented*?" The poor author of X, reputation lost, career shattered, will not dare show up in a software gathering for the next twenty years.) To avoid this pitfall, analysis methods have tended to err on the side of the clouds, relying on formalisms that do a good job of capturing overall structures, often through cloud-like graphical notations, but are quite limited when it comes to expressing the *semantic* properties of systems as required to address goal A2 (answering precise questions).

Many of the traditional analysis methods fit this description. Their success comes from their ability to list the components of a system and describe their relations graphically, making them the software equivalent of the *block diagrams* of other engineering disciplines. But they are not too good at capturing the semantics. For software projects this carries a risk: believing that you have completed a successful analysis when all you have really done is to define the major components and their relations, leaving out many deeper properties of the specification that may turn out to be critical.

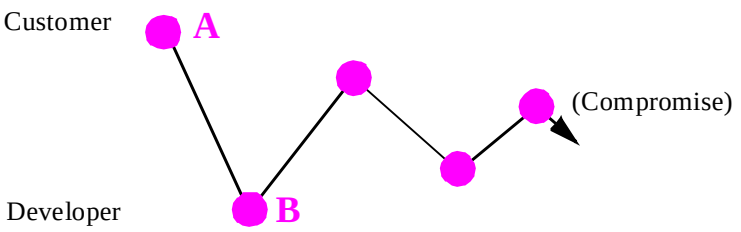
Later in this chapter we will study ideas for reconciling the goals of structural description and semantic precision.

27.2 THE CHANGING NATURE OF ANALYSIS

Although the object-oriented analysis literature hardly mentions this point, the most significant contribution of object technology to analysis is not technical but organizational. Object technology does not just provide new ways of doing analysis; it affects the very nature of the task and its role in the software process.

This change follows from the method's emphasis on reusability. If instead of assuming that every new project must start from scratch, considering the customer's requirements as the Gospel, we bring into the picture the presence of a regularly growing repertory of software components, some obtained (or obtainable) from the outside and some developed as a result of in-house projects, the process becomes different: not the execution of an order from above, but a **negotiation**.

Requirements analysis as a negotiation



The figure suggests this process: the customer starts with a requirement at **A**; you counter with a proposal at **B**, covering perhaps only part of the requirements, or a slightly different form of the requirements, but based for a large part on existing reusable components and hence achievable at significantly less cost and sooner. The customer may initially find the sacrifice of functionality too large; this opens a haggling phase which should eventually lead to an acceptable compromise.

The haggling has always been there, of course. The customer's requirements were the Gospel only in some descriptions of the "software process" in the software engineering literature, presenting an ideal view for pedagogical purposes, and perhaps in some government contracts. In most normal situations, the developers had some freedom to discuss requirements. But with the advent of object technology this officious phenomenon becomes an official part of the software development process, and gains new prominence with the development of reusable libraries.

27.3 THE CONTRIBUTION OF OBJECT TECHNOLOGY

Object technology also affects, of course, the techniques of analysis.

Here the most important thing to learn is that we have almost nothing to learn. The framework defined in the preceding chapters has more than enough to get us started with modeling. "More than enough" actually means too much: the notation includes an *operational* part, made of two components which we do not need for analysis:

- Instructions (assignments, loops, procedure calls, …) and all that goes with them.
- Routine bodies of the **do** form (but we do need **deferred** routines to specify operations without giving out their implementation).

If we ignore these imperative elements, we have a powerful system modeling method and notation. In particular:

- **Classes** will enable us to organize our system descriptions around object types, in the broad sense of the word "object" defined in preceding chapters (covering not just physical objects but also important concepts of the application domain).
- The **ADT** approach — the idea of characterizing objects by the applicable operations and their properties — yields clear, abstract, evolutionary specifications.
- To capture inter-component relations, the two basic mechanisms of "client" and inheritance are appropriate. The **client** relation, in particular, covers such information modeling concepts as "part of", association and aggregation.
- As we saw in the discussion of objects, the distinction between **reference** and **expanded** clients corresponds to the two basic kinds of modeling association.
- **Inheritance** — single, multiple and repeated — addresses classification. Even such seemingly specialized inheritance mechanisms as renaming will be precious to model analysis concepts.
- **Assertions** are essential to capture what was called above the *semantics* of systems: properties other than structural. Design by Contract is a powerful guide to analysis.
- **Libraries** of reusable classes will provide us — especially through their higher-level deferred classes — with ready-made specification elements.

This does not necessarily mean that the approach seen so far covers all the needs of system analysis (a question that will be discussed further below); but it certainly provides the right basis. The following example will provide some evidence.

27.4 PROGRAMMING A TV STATION

Let us see concretely how to apply the O-O concepts that we know to pure modeling.

The example involves organizing the schedule of a television station. Because it is drawn from a familiar application area, we can start it (although we most likely could not complete it) without the benefit of input from "domain experts", future users etc.; we can just, for the analysis exercise, rely on every layperson's understanding of TV.

Although the effort may be the prelude to the construction of a computerized system to manage the station's programming automatically, this possibility is neither certain nor relevant here; we are just interested in modeling.

Schedules

We concentrate on the schedule for a 24-hour period; the class (data abstraction) *SCHEDULE* presents itself. A schedule contains a sequence of individual program segments; let us start with

```
class SCHEDULE feature
    segments: LIST [SEGMENT]
end
```

When doing analysis we must constantly watch ourselves for fear of lapsing into overspecification. Is it overspecifying to use a *LIST*? No: *LIST* is a deferred class, describing the abstract notion of sequence; and television programming is indeed sequential, since one cannot broadcast two segments on the same station at the same time. By using *LIST* we capture a property of the problem, not the solution.

Note in passing the importance of reusability: by using classes such as *LIST* you immediately gain access to a whole set of features describing list operations: commands such as *put* for adding elements, queries such as the number of elements *count*. Reusability is as central to object-oriented analysis as it is to other O-O tasks.

What would be overspecifying here would be to *equate* the notion of schedule with that of list of segments. Object technology, as you will remember from the discussion of abstract data types, is implicit; it describes abstractions by listing their properties. Here there will certainly be more to a schedule than the list of its segments, so we need a separate class. Some of the other features of a schedule present themselves naturally:

See "More on implicitness", page 149.

```
indexing
    description: "Twenty-four hour TV schedules"
deferred class SCHEDULE feature
    segments: LIST [SEGMENT] is
            -- The successive segments
        deferred
        end
    air_time: DATE is
            -- Twenty-four hour period for this schedule
        deferred
        end
    set_air_time (t: DATE) is
            -- Assign this schedule to be broadcast at time t.
        require
            t.in_future
        deferred
        ensure
            air_time = t
        end
    print is
            -- Print paper version of schedule.
        deferred
        end
end
```

See "Using assertions for documentation: the short form of a class", page 389.

Note the use of deferred bodies. This is appropriate since by nature an analysis document is implementation-independent and even design-independent; having no body, deferred features are the proper tool. You could, of course, dispense with writing the **deferred** specification and instead use a formalism such as that of short forms. But two important arguments justify using the full notation:

- By writing texts that conform to the syntax of the software notation, you can make use of all the tools of the supporting software development environment. In particular, the compiling mechanism will double up as a precious CASE (computer-aided software engineering) tool, applying type rules and other validity constraints to check the consistency of your specifications and detect contradictions and ambiguities; and the browsing and documentation facilities of a good O-O environment will be as useful for analysis as they are for design and implementation.

- Using the software notation also means that, should you decide to proceed to the design and implementation of a software system, you will be able to follow a smooth transition path; your work will be to add new classes, effective versions of the deferred features and new features. This supports the *seamlessness* of the approach, discussed in the next chapter.

The class assumes a boolean query *in_future* on objects of type *DATE*; it only allows setting air time for future dates. Note our first use of a precondition and postcondition to express semantic properties of a system during analysis.

Segments

Rather than continuing to refine and enhance *SCHEDULE*, let us at this stage switch to the notion of *SEGMENT*. We can start with the following features:

```
indexing
        description: "Individual fragments of a broadcasting schedule"
deferred class SEGMENT feature
        schedule: SCHEDULE is deferred end
                -- Schedule to which segment belongs
        index: INTEGER is deferred end
                -- Position of segment in its schedule
        starting_time, ending_time: INTEGER is deferred end
                -- Beginning and end of scheduled air time
        next: SEGMENT is deferred end
                -- Segment to be played next, if any
        sponsor: COMPANY is deferred end
                -- Segment's principal sponsor
```

```
        rating: INTEGER is deferred end
                -- Segment's rating (for children's viewing etc.)
        … Commands such as change_next, set_sponsor, set_rating omitted …
        Minimum_duration: INTEGER is 30
                -- Minimum length of segments, in seconds
        Maximum_interval: INTEGER is 2
                -- Maximum time between two successive segments, in seconds
    invariant
        in_list: (1 <= index) and (index <= schedule.segments.count)
        in_schedule: schedule.segments.item (index) = Current
        next_in_list: (next /= Void) implies (schedule.segments.item (index + 1) = next)
        no_next_iff_last: (next = Void) = (index = schedule.segments.count)
        non_negative_rating: rating >= 0
        positive times: (starting_time > 0) and (ending_time > 0)
        sufficient_duration: ending_time – starting_time >= Minimum_duration
        decent_interval: (next.starting_time) – ending_time <= Maximum_interval
    end
```

Each segment "knows" the schedule of which it is a part, expressed by the query *schedule*, and its position in that schedule, expressed by *index*. It has a *starting_time* and an *ending_time*; we could also add a query *duration*, with an invariant clause expressing its relation to the previous two. Redundancy is acceptable in system analysis provided redundant features express concepts of interest to users or developers, and the relations between redundant elements are stated clearly through the invariant. Here, clauses *in_list* and *in_schedule* of the invariant express the relation between a segment's own *index* and its position in the schedule's list of segments.

A segment also knows about the segment that will follow, *next*. Invariant clauses again express the consistency requirements: clause *next_in_list* indicates that if the segment is at position *i* the *next* one is at position *i +1*; clause *no_next_iff_last*, that there is a *next* if and only if the segment is not the last in its schedule.

The last two invariant clauses express constraints on durations: *sufficient_duration* defines a minimum duration of 30 seconds for a program fragment to deserve being called a segment, and *decent_interval* a maximum of two seconds for the time between two successive segments (when the TV screen may go blank).

The class specification has taken two shortcuts that would almost certainly have to be removed at the next iteration of the analysis process. First, times and durations have been expressed as integers, measured in seconds; this is not abstract enough, and we should be able to rely on library classes *DATE*, *TIME* and *DURATION*. Second, the notion of *SEGMENT* covers two separate notions: a TV program fragment, which can be defined independently of its scheduling time; and the scheduling of a certain program at a certain time slot. To separate these two notions is easy; just add to *SEGMENT* an attribute

```
content: PROGRAM_FRAGMENT
```

with a new class *PROGRAM_FRAGMENT* describing the content independently of its scheduling. Feature *duration* should then appear in *PROGRAM_FRAGMENT*, and a new invariant clause of *SEGMENT* should state

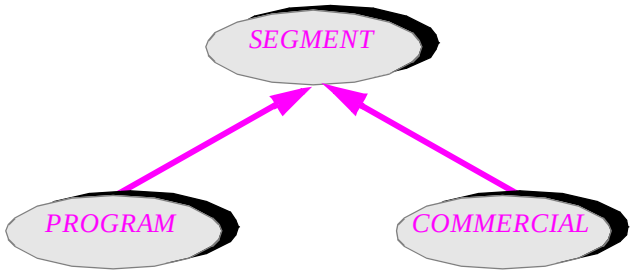
content.duration = ending_time – starting_time

For brevity the rest of this sketch continues to treat the content as part of the segment. Such discussions are typical of what goes on during the analysis process, aided by the object-oriented method: we examine various abstractions, discuss whether they justify different classes, move features to other classes if we think they have been misassigned.

A segment has a primary sponsor, and a rating. Although here too we might benefit from a separate class, *rating* has just been specified as an integer, with the convention that a higher rating implies more restrictions; 0 means a segment accessible to all audiences.

Programs and commercials

Probing the notion of *SEGMENT* further, we distinguish two kinds: program segments and commercial breaks (advertizing segments). This immediately suggests using inheritance:



See "TAXOMANIA", 24.4, page 820.

This urge to use inheritance during analysis, by the way, is always suspect; you should be wary of bouts of taxomania, prompting you to create spurious classes where simple distinctive properties would suffice. The guiding criterion was given in the description of inheritance: does each proposed class really correspond to a separate abstraction, characterized by specific features and properties? Here the answer will be yes; it is not difficult to think of features for both programs and commercials, as will be listed in part below. Using inheritance will also yield the benefit of openness: we can add a new heir such as *INFOMERCIAL* later to describe segments of a different kind.

We can start *COMMERCIAL* as follows:

```
indexing
        description: "Advertizing segment"
deferred class COMMERCIAL inherit
        SEGMENT
                rename sponsor as advertizer end
```

```
feature
        primary: PROGRAM is deferred
                -- Program to which this commercial is attached
        primary_index: INTEGER is deferred
                -- Index of primary
        set_primary (p: PROGRAM) is
                        -- Attach commercial to p.
                require
                        program_exists: p /= Void
                        same_schedule: p.schedule = schedule
                        before: p.starting_time <= starting_time
                deferred
                ensure
                        index_updated: primary_index = p.index
                        primary_updated: primary = p
                end
invariant
        meaningful_primary_index: primary_index = primary.index
        primary_before: primary.starting_time <= starting_time
        acceptable_sponsor: advertizer.compatible (primary.sponsor)
        acceptable_rating: rating <= primary.rating
end
```

Note the use of renaming, another example of a notational facility that at first sight might have appeared to be useful mostly for implementation-level classes, but turns out to be just as necessary for modeling. When a segment is a commercial, it is more appropriate to refer to its *sponsor* as being its *advertizer*.

Every commercial segment is attached to an earlier program segment (not a commercial), its *primary*, whose index in the schedule is *primary_index*. The first two invariant clauses express consistency conditions; the last two express compatibility rules:

- If a show has a sponsor, any advertizer during that show must be acceptable to it; you do not advertize for Pepsi-Cola during a show sponsored by Coca-Cola. The query *compatible* of class *COMPANY* might be given through some database.
- The rating of a commercial must be compatible with that of its primary program: you should not advertize for *Bulldozer Massacre III* on a toddlers' program.

The notion of *primary* needs refinement. It becomes clear at this stage of our analysis that we should really add a level: instead of a schedule being a succession of program segments and commercials, we should view it as a succession of shows, where each show (described by a class *SHOW*) has its own features, such as the show's sponsor, and a succession of show segments and commercials. Such improvement and refinement, developed as we gain more insight into the problem and learn from our first attempts, are a normal component of the analysis process.

Business rules

We have seen how invariant clauses and other assertions can cover semantic constraints of the application domain, also known in analysis parlance as *business rules*: in class *SCHEDULE*, that one can schedule a segment only in the future; in *SEGMENT*, that the interruption between two segments may not exceed a preset duration; in *COMMERCIAL*, that a commercial's rating must be compatible with that of the enclosing program.

It is indeed one of the principal contributions of the method that you can use assertions and the principles of Design by Contract to express such rules along with the structure, avoiding both the clouds and the precipice.

A practical warning however: even without any implementation commitment, there is a risk of overspecification. In assertions of the analysis text, you should only include business rules that have a high degree of certainty and durability. If any rule is subject to change, use abstraction to express what you need but leave room for adaptation. For example the rules on sponsor-advertizer compatibility can change; so the invariant of *COMMERCIAL* stays away from overspecification by simply postulating a boolean-valued query *compatible* in class *COMPANY*. One of the great advantages of analysis is that you choose what you say and what you say not. State what is known — if you specify nothing, the specification will not be of much interest — but no more. This is the same comment that we encountered in the discussion of abstract data types: we want the truth, all the relevant truth, but nothing *more* than the truth.

> That ADT comments should be directly applicable here is no surprise: ADTs are a high-level specification technique, and in fact the use of deferred classes with their assertions as a tool for analysis, illustrated by the TV station example, is conceptually a variant of ADT specification using software syntax.

Assessment

Although we have only begun the TV station programming example, we have gone far enough to understand the general principles of the approach. What is striking is how powerful and intuitive the concepts and notation are for general, software-independent system modeling, even though they were initially developed (in earlier chapters) for software purposes and, to the superficial observer, may even appear to address just *programming* issues. Here they come out in their full scope: as a general-purpose method and notation for describing systems of many kinds, covering the structure of systems as well as fine aspects of their semantics, and able to tackle complexity as well as evolution.

Nothing in a specification of the kind illustrated above is implementation-related, or even software-related, or even computer-related. We are using the concepts of object technology for purely descriptive purposes; no computer need enter the picture.

Of course if you or your customer do decide to go ahead and build a software system for managing TV station programming, you will have the tremendous advantage of a description that is already in a software-like form, syntactically and structurally. The transition to a design and implementation will proceed seamlessly in the same framework; you may even be able to retain many of the analysis classes *as is* in the final system, with implementations provided in proper descendants.

27.5 EXPRESSING THE ANALYSIS: MULTIPLE VIEWS

The use of specifications expressed in a software-like language, illustrated by the TV station example, raises an obvious question of practicality in normal industrial environments.

What can cause some skepticism is that the people who will have to review the analysis document may not all be comfortable with such notations; more than any other stage, analysis is the time for collaboration with application domain experts, future users, managers, contract administrators. Can we expect to them to read a specification that at first sight looks like a software text (although it is a pure model), and possibly contribute to it?

Surprisingly often, the answer is yes. Understanding the part of the notation that serves for analysis, as illustrated by the preceding example, does not require in-depth software expertise, simply an understanding of elements of the basic laws of logic and organized reasoning in any discipline. I can attest to having used such specifications successfully with people of widely different backgrounds and education.

But this is not the end of the story. A core of formalism-averse people may remain, whose input you will still need. And even those who appreciate the power of the formalism will need other views, in particular graphical representations. In fact the recurrent fights about graphics versus formalism, formalism versus natural language, are pointless. In practice the description of a non-trivial system requires **several** complementary views, such as:

- A formal text, as illustrated in the preceding example.
- A graphical representation, showing system structures in terms of "bubble and arrow" diagrams (also used in one instance for the example). Here the graphs will show classes, clusters, objects, and relations such as client and inheritance.
- A natural-language requirements document.
- Perhaps a tabular form, as appears in the presentation of the BON method below.

Each such view has its unique advantages, addressing some of the multiple goals of analysis defined at the beginning of this chapter; each has limitations that may make it irrelevant to other goals. In particular:

- Natural-language documents are irreplaceable for conveying essential ideas and explaining fine nuances. But they are notoriously prone to imprecision and ambiguity, as we saw in the critique of the "underline the nouns" approach.

"STUDYING A REQUIREMENTS DOCUMENT", 22.1, page 720.

- Tabular representations are useful to collect a set of related properties, such as the principal characteristics of a class — parents, features, invariant.
- Graphical representations are excellent for describing *structural* properties of a problem or system by showing the components and their relations. This explains the success of "bubble-and-arrow" descriptions as promoted by "structured analysis". But they are severely limited when it comes to expressing precise *semantic* properties, as required by item A3 of the list of analysis goals (answering specific

questions). For example a graphical description is not the best place to look at for an answer to the question "*what is the maximum length of a commercial break*?".

- Formal textual representations, such as the notation of this book, are the best tool for answering such precise questions, although they cannot compete with graphical representation when the goal is simply to get a quick understanding of how a system is organized.

The usual argument for graphical representations over textual ones is the cliché that "a picture is worth a thousand words". It has its share of truth; block diagrams are indeed unsurpassed to convey to the reader the overall impression of a structure. But the proverb conveniently ignores the details that the words can carry, the imprecision that can affect the picture, and the *errors* that it can contain. The next time someone invites you to use a diagram as the final specification of some delicate aspect of a system, look at the comics page of the daily paper: the "find the differences between these two variants" teasers do not ask you to rack your eyes and brain over two sentences or two paragraphs, but to find the hidden differences between two deceptively similar *pictures*.

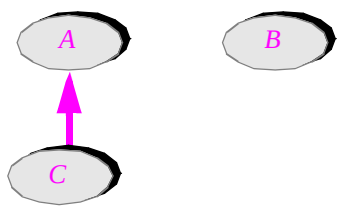
So what we need with a good analysis method is a way to use each one of these views as the need arises, switching freely from one to the other.

The question then arises of how to maintain consistency between the various views. The solution is to use one of the views as the reference, and to rely on software tools to make sure that additions and changes get propagated to all views. The best candidate to serve as reference — the only credible one, in fact — is the formal text, precisely because it is the only one that is both defined rigorously and able to cover semantics as well as structural properties.

With this approach, the use of formal software-like descriptions is not exclusive of other styles, and you can use a variety of tools adapted to the expertise levels and personal tastes of the analysis participants (software people, managers, end users). For the formal text, the software development environment may be appropriate: we have seen in particular that the compiler can double as an analysis support tool thanks to its facilities for checking type rules and other validity constraints, although its code generation mechanism is irrelevant at this stage. For the graphical notation, you will use a graphical CASE tool, apt at producing and manipulating structure charts. For the natural language texts, document manipulation and management systems can help. Tables can also have specific tool support. The various tools involved can be either separate or integrated in an analysis or development workbench.

Graphical or tabular input will immediately be reflected in the formal representation; for example if the graphical view showed a class *C* inheriting from a class *A*

Inheritance link



and you interactively redirect the arrow to point to *B*, the tools will automatically change the **inherit** clause of the formal text to reflect the change. Conversely, if you edit the formal description, the graphical and tabular representations will be updated.

It is more difficult for tools to process changes in natural-language descriptions. But if the document manipulation system enforces structured system descriptions, with chapters, sections and paragraphs, it is possible to keep links between the formal text and the natural-language requirements document, for example to indicate that a certain class or feature is connected to a certain paragraph of the requirements; this is particularly useful when the environment also provides configuration management tools, so that when something changes in the requirements the tools can, if not update the formal description, at least alert you to the change and produce a list of all the elements that depended, directly or indirectly, on the modified part.

The other direction is interesting too: producing natural-language descriptions from formal texts. The idea is simply to reconstruct, from a formal system description, a natural-language text that would express the same information in a form that will not scare the more formalism-averse members of the target readership. It is not hard indeed to think of a tool that, starting from our analysis sketch, would produce a fake English form such as

1. System concepts

The concepts of this system are:

SCHEDULE, SEGMENT, COMMERCIAL, PROGRAM …

SCHEDULE is discussed in section 2; *SEGMENT* is discussed in section 3; [etc.]

2. The notion of *SCHEDULE*

…

3. …

4. The notion of *COMMERCIAL*

4.1 General description:

Advertizing segments

4.2 Source notions.

The notion of *COMMERCIAL* is a specialized case of the notion of *SEGMENT* and has all its operations and properties, except for redefined ones as listed below.

4.2 Renamed operations.

What is called *sponsor* for *SEGMENT* is called *advertizer* for *COMMERCIAL*.

…

4.3 Redefined operations

…

4.4 New operations

The following operations characterize a *COMMERCIAL*:

primary, a query returning a *PROGRAM*

Needs: none [Arguments, if any, would be listed here]

Description:

Program to which commercial is attached

Input conditions:

…
Result conditions:
…
… Other operations …
4.5 Constraints
… An English-like rendition of the invariant properties …
4. The notion of *PROGRAM*
…
etc.

All the English sentences ("The concepts of this system are", "The following operations characterize a …" and so on) are drawn from a standard set of predefined formulae, so they are not really "natural" language; but the illusion can be strong enough to make the result palatable to non-technical people, with the guarantee that it is consistent with the more formal view since it has been mechanically derived from it.

Although I do not know any tool that has explored this idea very far, the goal seems reachable. A project to build such a tool would be several orders of magnitude more realistic than long-going efforts in the reverse direction (attempts at automatic *analysis* of natural-language requirements documents) which have never been able to produce much, because of the inherent difficulty of analyzing natural language. Here we are interested in natural language *generation*, an easier task (in the same way that speech synthesis has progressed faster than speech recognition).

What makes this possible is the generality of the formal notation and, especially, its support for assertions, allowing us to include useful semantic properties in the generated natural-language texts. Without assertions we would remain in the vague — in the clouds.

27.6 ANALYSIS METHODS

Here is a list of some of the best-known methods of O-O analysis, listed in the approximate order of their public appearance. Although the description focuses on the analysis component of the methods, note that most of them also include design-related or even implementation-related components. The short summaries cannot do justice to the methods; to learn more, see the books and Web pages listed in the bibliographic notes to this chapter.

The **Coad-Yourdon** method initially resulted from an effort to objectify ideas coming from structured analysis. It involves five stages: finding classes and objects, starting from the application domain and analyzing system responsibilities; identifying structures by looking for generalization-specialization and whole-part relationships; defining "subjects" (class-object groups); defining attributes; defining services.

The **OMT** method (Object Modeling Technique) combines concepts of object technology with those of entity-relation modeling. The method includes a static model, based on the concepts of class, attribute, operation, relation and aggregation, and a dynamic model based on event-state diagrams, describing in an abstract way the intended behavior of the system.

The **Shlaer-Mellor** method is original in its emphasis on producing models that lend themselves to simulation and execution, making it possible to validate model behavior independently of any design or implementation. To separate concerns, it divides the problem into a number of domains: application domain, service domains (such as the user interface domain), software architecture domain, implementation domains (such as operating system or language). Rather than seamless development, its model for the development process uses translation to link the domains together into code for final system construction.

> The presence of architecture, design and implementation models in Shlaer-Mellor and some of the following methods illustrates the comment made above that the methods' ambition often extends beyond analysis to cover a large part of the lifecycle, or all of it.

In the **Martin-Odell** method, also known as OOIE (Object-Oriented Information Engineering), analysis consists of two parts: object structure analysis, which identifies the object types and their composition and inheritance relations; and object behavior analysis, which defines the dynamic model by considering object states and the events that may change these states. The events are considered first, leading to the identification of classes.

The **Booch** method uses a logical model (class and object structure) and a physical model (module and process architecture), including both static and dynamic components, and relying on numerous graphical symbols. It is intended to be subsumed by the "Unified Modeling Language" (see below).

The **OOSE** method (Object-Oriented Software Engineering), also known as Jacobson's method or as Objectory, the name of the original supporting tool, relies on use cases (scenarios) to elicit classes. It distinguishes five use case models: domain object model, analysis model (the use cases structured by the analysis), design model, implementation model, testing model.

See "Use cases", page 738.

The **OSA** method (for Object-oriented Systems Analysis) is meant to provide a general model of the analysis process rather than a step-by-step procedure. It consists of three parts: the object-relationship model, which describes objects and classes as well as their relations — with each other and with the "real world"; the object-behavior model, which provides the dynamic view through states, transitions, events, actions and exceptions; and the object-interaction model, specifying possible interactions between objects. The method also supports a notion of view, as well as generalization and specialization, which apply to both the interaction and behavior models.

The **Fusion** method seeks to combine some of the best ideas of earlier methods. For analysis it includes an object model, devoted to the problem domain, and an interface model, describing system behavior. The interface model is itself made of an operation model, specifying events and the resulting operations, and a lifecycle model, describing scenarios that guide the evolution of the system. Analysts should maintain a data dictionary which collects all the information from the various models.

The **Syntropy** method defines three models: the essential model "*is a model of a real or imaginary situation*, [*having nothing*] *to do with software*: *it describes the elements of the situation, their structure and behavior*". The specification model is an abstract model that treats the system as a stimulus-response mechanism, assuming unlimited hardware resources. The implementation model takes into account the actual computing environment. Each model may be expressed along several views: a type view describing object types and their static properties; state views, similar to the state transition diagrams of OMT, to

Citation from the Syntropy Web page listed in the bibliography section.

describe dynamic behavior; and mechanisms diagrams for implementation. The method also supports a notion of viewpoint to describe various interfaces to the same objects, going beyond the mere separation of interface and implementation provided by O-O languages.

The **MOSES** method involves five models: object-class; event, showing class collaboration by describing what messages are triggered as a result of calling a service on an object; "objectcharts", to model state-transition dynamics; inheritance; and service structure, to show data flow. Like the Business Object Notation reviewed in the next section, MOSES emphasizes the importance of contracts in specifying a class, using preconditions, postconditions and invariants in the style of the present book. Its "fountain" process model defines a number of standard documents to be produced at each stage.

The **SOMA** method (Semantic Object Modeling Approach) uses a "Task Object Model" to capture the requirements and transforms them into a "Business Object Model". It is one of the few methods to have benefited from formal approaches, using a notion of contract to describe business rules applying to objects.

At the time of writing, two separate efforts are progressing to unify existing methods. One, led by Brian Henderson-Sellers, Don Firesmith, Ian Graham and Jim Odell, is intended to produce an OPEN (the retained name) unified method. The other, by Rational Corporation, is starting from the OMT, Booch and Jacobson methods to define a "Unified Modeling Language".

27.7 THE BUSINESS OBJECT NOTATION

Each of the approaches listed in the preceding sections has its strong points. The method that seems to provide the most benefit for the least complexity is Nerson's and Waldén's Business Object Notation; let us take a slightly closer look at it to gain some insight into what a comprehensive approach to O-O analysis requires. This brief presentation will only sketch the principal features of the method, limiting itself to its contribution to analysis; for more details, and to explore design and implementation aspects, see the Waldén-Nerson book cited in the bibliography.

The Business Object Notation started as a graphical formalism for representing system structures. The original name was kept, even though BON has grown from just a notation to a complete development method. BON has been used in many different application areas for the analysis and development of systems, some very complex.

BON is based on three principles: *seamlessness*, *reversibility* and *contracting*. Seamlessness is the use of a continuous process throughout the software lifecycle. Reversibility is the support for both forward and backward engineering: from analysis to design and implementation, and back. Contracting (remember *Design by Contract*) is the precise definition, for each software element, of the associated semantic properties; BON is almost the only one among the popular analysis methods to use a full-fledged assertion mechanism, allowing analysts to specify not only the structure of a system but also its semantics (constraints, invariants, properties of the expected results).

Several other properties make BON stand out among O-O methods:

- It is meant to "scale up", in the sense explained at the beginning of this chapter. Various facilities and conventions enable you to choose the level of abstraction of a

system or subsystem description, to zoom in on a component, to hide parts of a description. This selective hiding is preferable, in my opinion, to the use of multiple models illustrated by some of the preceding methods: here, for seamlessness and reversibility, you keep a single model; but you can at any time decide what aspects are relevant to your needs of the moment, and hide the rest.

- BON, created in the nineteen-nineties, was designed under the assumption that its users would have access to computing resources, not just paper and whiteboards. This makes it possible to use powerful tools to display complex information, free from the tyranny of fixed-size areas such as paper pages. Such a tool is sketched in the last chapter of this book. For small examples, the method can of course be used with pencil and paper.
- For all its ambition, especially its ability to cover large and complex systems, the method is notable for its simplicity. It only involves a small number of basic concepts. Note in particular that the formalism can be described over two pages; the most important elements appear below and on the facing page.

BON's support for large systems relies in part on the notion of **cluster**, denoting a group of logically related classes. Clusters can include subclusters, so that the result is a nested structure allowing analysts to work on various levels at different times. Some of the clusters may of course be libraries; the method puts a strong emphasis on reuse.

For further discussion of clusters see "CLUSTERS", 28.1, page 923.

The static part of the model focuses on classes and clusters; the dynamic part describes objects, object interactions and possible scenarios for message sequencing.

BON recognizes the need for several complementary formalisms, explained earlier in this chapter. (The assumed availability of software tools is essential here: with a manual process, multiple views would raise the issue of how to maintain the *consistency* of the model; tools can ensure it automatically.) The formalisms include a textual notation, a tabular form and graphical diagrams.

The *textual notation* is similar to the notation of this book; but since it does not have to be directly compilable, it can use a few extensions in the area of assertions, including **delta** *a* to specify that a feature can change an attribute *a*, **forall** and **exists** to express logic formulae of first-order predicate calculus, and set operators such as **member_of**.

The *tabular form* is convenient to summarize the properties of a class compactly. Here is the general form of a tabular class chart:

CLASS	*Class_name*		Part:
Short description		Indexing information	
Inherits from			
Queries			
Commands			
Constraints			

"Constraints" are invariants.

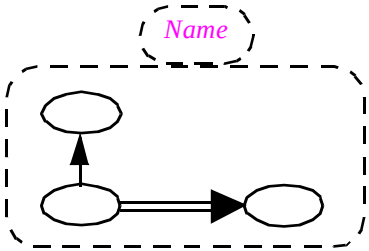
The graphical notation is extremely simple, so as to be easy to learn and remember. The principal conventions, static as well as dynamic, appear below.

Main diagram types of the Business Object Notation

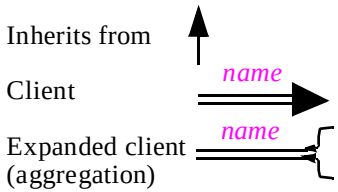
(*After* [Waldén 1995], *used with permission*.)

STATIC DIAGRAMS

Cluster (with some classes)

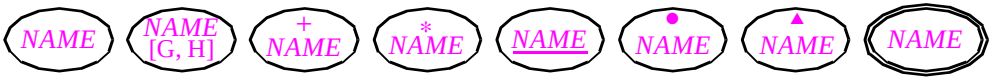


Inter-class relations

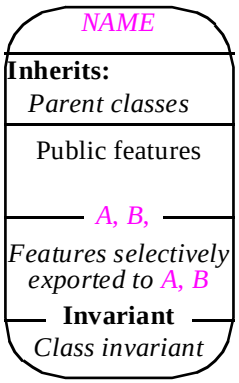


Multiplicity of relations

Class: generic, effective, deferred, reused, persistent, interfaced, root.



Class: detailed interface



Features

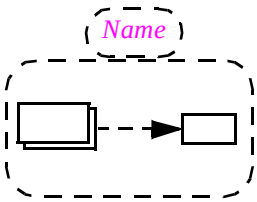
*name**, *name*$^{+}$, *name*$^{++}$	deferred, effective, redefined
→ *name*: *TYPE*	input argument
? !	precondition, postcondition

Assertion operators

Δ *name*	feature may change attribute *name*
@, ýýý∅	current object, void reference
∃, ∀, |, •	symbols for predicate calculus operations
∈ , ∉	membership operators

DYNAMIC DIAGRAMS

Object group (with some objects)



Object *Name*

Objects (one or more) *Name*

Inter-object relations

Message passing (with message number from scenario) 7

The method defines a precise process for analysis and development, consisting of seven tasks. The order of tasks corresponds to an ideal process, but the method recognizes that in practice it is subject to variation and iteration, as implied in fact by the very concept of reversibility. The standard tasks are:

B1 • **Delineate system borderline**: identify what the system will include and not include; define major subsystems, user metaphors, functionality, reused libraries.

B2 • **List candidate classes**: produce first list of classes based on problem domain.

B3 • **Select classes and group into clusters**: organize classes in logical groups, decide what classes will be deferred, persistent, externally interfaced etc.

B4 • **Define classes**: expand the initial definition of classes to specify each of them in terms of queries, commands and constraints.

B5 • S**ketch system behavior**: define charts for object creation, events and scenarios.

B6 • **Define public features**: finalize class interfaces.

B7 • **Refine system**.

Throughout the process, the method prescribes keeping a **glossary** of terms of the technical domain. Experience shows this to be an essential tool for any large application project, both to give non-experts a place to go when they do not understand some of the domain experts' jargon, and to make sure that the experts actually agree on the terms (it is surprising to see how often the process reveals that they do not!).

More generally, the method specifies for each step is a precise list of its deliverables: documents that the manager is entitled to expect as a result of the step's work. This precision in defining organizational responsibilities makes BON not only an analysis and design method but also a strategic tool for project management.

27.8 BIBLIOGRAPHY

The principal reference on the Business Object Notation is [Waldén 1995]. The basic concepts were introduced in [Nerson 1992]. A Web page is available at *www.tools.com/products/bon/*.

Add the ritual ***http://*** *as a prefix to all Web addresses.*

Here are the principal references on other methods, with associated Web addresses. Coad-Yourdon: [Coad 1990], *www.oi.com*; OMT: [Rumbaugh 1991]; Shlaer-Mellor [Shlaer 1992], *www.projtech.com*; Martin-Odell, [Martin 1992]; Booch: [Booch 1994]; OOSE: [Jacobson 1992]; OSA: [Embley 1992], *osm7.cs.byu.edu/OSA.html*; Syntropy: [Cook 1994], *www.objectdesigners.co.uk/syntropy*; Fusion, [Coleman 1994]; MOSES: [Henderson-Sellers 1994], *www.csse.swin.edu.au/cotar/OPEN/OPEN.html*; SOMA, [Graham 1995].

On the OPEN method convergence project see [Henderson-Sellers 1996]; [Computer 1996] is a discussion of Rational's Unified Modeling Language effort (Booch-OMT-Jacobson).

Katsuya Amako maintains a set of descriptions of O-O methods, along with other useful O-O information, at *arkhp1.kek.jp/~amako/OOInfo.html*.