CHAPTER 1

HISTORY AND MOTIVATION

The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.

— Donald Knuth (19??)

How could anyone diligently concentrate on his work on an afternoon with such warmth, splendid sunshine, and blue sky. This rhetorical question was one I asked many times while spending a sabbatical leave in California in 1985. Back home everyone would feel compelled to profit from the sunny spells to enjoy life at leisure in the country-side, wandering or engaging in one's favourite sport. But here, every day was like that, and giving in to such temptations would have meant the end of all work. And, had I not chosen this location in the world because of its inviting, enjoyable climate?

Fortunately, my work was also enticing, making it easier to buckle down. I had the privilege of sitting in front of the most advanced and powerful work-station anywhere, learning the secrets of perhaps the newest fad in our fast developing trade, pushing colored rectangles from one place of the screen to another. This all had to happen under strict observance of rules imposed by physical laws and by the newest technology. Fortunately, the advanced computer would complain immediately if such a rule was violated, it was a rule checker and acted like your big brother, preventing you from making steps towards disaster. And it did what would have been impossible for oneself, keeping track of thousands of constraints among the thousands of rectangles laid out. This was called computer-aided design. "Aided" is rather a euphemism, but the computer did not complain about the degradation of its role.

While my eyes were glued to the colorful display, and while I was confronted with the evidence of my latest inadequacy, in through the always open door stepped my colleague (JG). He also happened to spend a leave from duties at home at the same laboratory, yet his face did not exactly express happiness, but rather frustration. The chocolate bar in his hand did for him what the coffee cup or the pipe does for others, providing temporary relaxation and distraction. It was not the first time he appeared in this mood, and without words I guessed its cause. And the episode would reoccur many times.

His days were not filled with the great fun of rectangle-pushing; he had an assignment. He was charged with the design of a compiler for the same advanced computer. Therefore, he was forced to deal much more closely, if not intimately, with the underlying software system. Its rather frequent failures had to be understood in his case, for he was programming, whereas I was only using

it through an application; in short, I was an end-user! These failures had to be understood not for purposes of correction, but in order to find ways to avoid them. How was the necessary insight to be obtained? I realized at this moment that I had so far avoided this question; I had limited familiarization with this novel system to the bare necessities which sufficed for the task on my mind.

It soon became clear that a study of the system was nearly impossible. Its dimensions were simply awesome, and documentation accordingly sparse. Answers to questions that were momentarily pressing could best be obtained by interviewing the system's designers, who all were in-house. In doing so, we made the shocking discovery that often we could not understand their language. Explanations were fraught with jargon and references to other parts of the system which had remained equally enigmatic to us.

So, our frustration-triggered breaks from compiler construction and chip design became devoted to attempts to identify the essence, the foundations of the system's novel aspects. What made it different from conventional operating systems? Which of these concepts were essential, which ones could be improved, simplified, or even discarded? And where were they rooted? Could the system's essence be distilled and extracted, like in a chemical process?

During the ensuing discussions, the idea emerged slowly to undertake our own design. And suddenly it had become concrete. "crazy" was my first reaction, and "impossible". The sheer amount of work appeared as overwhelming. After all, we both had to carry our share of teaching duties back home. But the thought was implanted and continued to occupy our minds.

Sometime thereafter, events back home suggested that I should take over the important course about System Software. As it was the unwritten rule that it should primarily deal with operating system principles, I hesitated. My scruples were easily justified: After all I had never designed such a system nor a part of it. And how can one teach an engineering subject without first-hand experience?

Impossible? Had we not designed compilers, operating systems, and document editors in small teams? And had I not repeatedly experienced that an inadequate and frustrating program could be programmed from scratch in a fraction of source code used by the original design? Our brain-storming continued, with many intermissions, over several weeks, and certain shapes of a system structure slowly emerged through the haze. After some time, the preposterous decision was made: we would embark on the design of an operating system for our workstation (which happened to be much less powerful than the one used for my rectangle-pushing) from scratch.

The primary goal, to personally obtain first-hand experience, and to reach full understanding of every detail, inherently determined our manpower: two part-time programmers. We tentatively set our time-limit for completion to three years. As it later turned out, this had been a good estimate; programming was begun in early 1986, and a first version of the system was released in the fall of 1988.

Although the search for an appropriate name for a project is usually a minor problem and often left to chance and whim of the designers, this may be the place

to recount how Oberon entered the picture in our case. It happened that around the time of the beginning of our effort, the space probe Voyager made headlines with a series of spectacular pictures taken of the planet Uranus and of its moons, the largest of which is named Oberon. Since its launch I had considered the Voyager project as a singularly well-planned and successful endeavor, and as a small tribute to it I picked the name of its latest object of investigation. There are indeed very few engineering projects whose products perform way beyond expectations and beyond their anticipated lifetime; mostly they fail much earlier, particularly in the domain of software. And, last but not least, we recall that Oberon is famous as the king of elfs.

The consciously planned shortage of manpower enforced a single, but healthy, guideline: Concentrate on essential functions and omit embellishments that merely cater to established conventions and passing tastes. Of course, the essential core had first to be recognized and crystallized. But the basis had been laid. The ground rule became even more crucial when we decided that the result should be able to be used as teaching material. I remembered C.A.R. Hoare's plea that books should be written presenting actually operational systems rather than half-baked, abstract principles. He had complained in the early 1970s that in our field engineers were told to constantly create new artifacts without being given the chance to study previous works that had proven their worth in the field. How right was he, even to the present day!

The emerging goal to publish the result with all its details let the choice of programming language appear in a new light: it became crucial. Modula-2 which we had planned to use, appeared as not quite satisfactory. Firstly, it lacked a facility to express extensibility in an adequate way. And we had put extensibility among the principal properties of the new system. By "adequate" we include machine-independence. Our programs should be expressed in a manner that makes no reference to machine peculiarities and low-level programming facilities, perhaps with the exception of device interfaces, where dependence is inherent.

Hence, Modula-2 was extended with a feature that is now known as type extension. We also recognized that Modula-2 contained several facilities that we would not need, that do not genuinely contribute to its power of expression, but at the same time increase the complexity of the compiler. But the compiler would not only have to be implemented, but also to be described, studied, and understood. This led to the decision to start from a clean slate also in the domain of language design, and to apply the same principle to it: concentrate on the essential, purge the rest. The new language, which still bears much resemblance to Modula-2, was given the same name as the system: Oberon [1] [2] In contrast to its ancestor it is terser and, above all, a significant step towards expressing programs on a high level of abstraction without reference to machine-specific features.

1 N. Wirth. The programming language Oberon. Software - Practice and Experience 18, 7, (July 1988) 671-690.

2 M. Reiser and N. Wirth. Programming in Oberon - Steps beyond Pascal and Modula. Addison- Wesley, 1992.

We started designing the system in late fall 1985, and programming in early 1986. As a vehicle we used our workstation Lilith and its language Modula-2. First, a cross-compiler was developed, then followed the modules of the inner core together with the necessary testing and down-loading facilities. The development of the display and the text system proceeded simultaneously, without the possibility of testing, of course. We learned how the absence of a debugger, and even more so the absence of a compiler, can contribute to careful programming.

Thereafter followed the translation of the compiler into Oberon. This was swiftly done, because the original had been written with anticipation of the later translation. After its availability on the target computer Ceres, together with the operability of the text editing facility, the umbilical cord to Lilith could be cut off. The Oberon System had become real, at least its draft version. This happened around the middle of 1987; its description was published thereafter 3, and a manual and guide followed in 1991 5.

The system's completion took another year and concentrated on connecting the workstations in a network for file transfer 4, on a central printing facility, and on maintenance tools. The goal of completing the system within three years had been met. The system was introduced in the middle of 1988 to a wider user community, and work on applications could start. A service for electronic mail was developed, a graphics system was added, and various efforts for general document preparation systems proceeded. The display facility was extended to accommodate two screens, including color. At the same time, feedback from experience in its use was incorporated by improving existing parts. Since 1989, Oberon has replaced Modula-2 in our introductory programming courses.

---

3 N. Wirth and J. Gutknecht. The Oberon System. Software - Practice and Experience, 19, 9 (Sept. 1989), 857-893.

5 M. Reiser. The Oberon System - User Guide and Programmer's Manual. Addison-Wesley, 1991.

4 N. Wirth. Ceres-Net: A low-cost computer network. Software - Practice and Experience, 20, 1 (Jan. 1990), 13-24.

CHAPTER 2

STRUCTURE OF THE SYSTEM

2.1. INTRODUCTION

In order to warrant the sizeable effort of designing and constructing an entire operating system from scratch, a number of basic concepts need to be novel. We start this chapter with a discussion of the principal concepts underlying the Oberon System and of the dominant design decisions. On this basis, a presentation of the system's structure follows. It will be restricted to its coarsest level, namely the composition and interdependence of the largest building blocks, the modules. The chapter ends with an overview of the remainder of the book. It should help the reader to understand the role, place, and significance of the parts described in the individual chapters.

The fundamental objective of an operating system is to present the computer to the user and to the programmer at a certain level of abstraction. For example, the store is presented in terms of requestable pieces or variables of a specified data type, the disk is presented in terms of sequences of characters (or bytes) called files, the display is presented as rectangular areas called viewers, the keyboard is presented as an input stream of characters, and the mouse appears as a pair of coordinates and a set of key states. Every abstraction is characterized by certain properties and governed by a set of operations. It is the task of the system to implement these operations and to manage them, constrained by the available resources of the underlying computer. This is commonly called resource management.

Every abstraction inherently hides details, namely those from which it abstracts. Hiding may occur at different levels. For example, the computer may allow certain parts of the store, or certain devices to be made inaccessible according to its mode of operation (user/supervisor mode), or the programming language may make certain parts inaccessible through a hiding facility inherent in its visibility rules. The latter is of course much more flexible and powerful, and the former indeed plays an almost negligible role in our system. Hiding is important because it allows maintenance of certain properties (called *invariants*) of an abstraction to be guaranteed. Abstraction is indeed the key of any modularization, and without modularization every hope of being able to guarantee reliability and correctness vanishes. Clearly, the Oberon System was designed with the goal of establishing a modular structure on the basis of purpose-oriented abstractions. The availability of an appropriate programming language is an indispensable prerequisite, and the importance of its choice cannot be over-emphasized.

2.2. CONCEPTS

2.2.1. Viewers.

Whereas the abstractions of individual variables representing parts of the primary store, and of files representing parts of the disk store are well established notions and have significance in every computer system, abstractions regarding input and output devices became important with the advent of high interactivity between user and computer. High interactivity requires high bandwidth, and the only channel of human users with high bandwidth is the eye. Consequently, the computer's visual output unit must be properly matched with the human eye. This occurred with the advent of the high-resolution display in the mid 1970s, which in turn had become feasible due to faster and cheaper electronic memory components. The high-resolution display marked one of the few very significant break-throughs in the history of computer development. The typical bandwidth of a modern display is in the order of 100 MHz. Primarily the high-resolution display made visual output a subject of abstraction and *resource management*. In the Oberon System, the display is partitioned into *viewers*, also called *windows*, or more precisely, into *frames*, rectangular areas of the screen(s). A viewer typically consists of two frames, a title bar containing a subject name and a menu of commands, and a main frame containing some text, graphic, picture, or other object. A viewer itself is a frame; frames can be nested, in principle to any depth.

The System provides routines for generating a frame (viewer), for moving and for closing it. It allocates a new viewer at a specified place, and upon request delivers hints as to where it might best be placed. It keeps track of the set of opened viewers. This is what is called *viewer management*, in contrast to the handling of their displayed contents.

But high interactivity requires not only a high bandwidth for visual output, it demands also flexibility of input. Surely, there is no need for an equally large bandwidth, but a keyboard limited by the speed of typing to about 100 Hz is not good enough. The break-through on this front was achieved by the so-called *mouse*, a pointing device which appeared roughly at the same time as the high-resolution display.

This was by no means just a lucky coincidence. The mouse comes to fruition only through appropriate software and the high-resolution display. It is itself a conceptually very simple device delivering signals when moved on the table. These signals allow the computer to update the position of a mark—the cursor —on the display. Since feedback occurs through the human eye, no great precision is required from the mouse. For example, when the user wishes to identify a certain object on the screen, such as a letter, he moves the mouse as long as required until the mapped cursor reaches the object. This stands in marked contrast to a digitizer which is supposed to deliver exact coordinates. The Oberon System relies very much on the availability of a mouse.

Perhaps the cleverest idea was to equip mice with buttons. By being able to signal a request with the same hand that determines the cursor position,

the user obtains the direct impression of issuing position-dependent requests. Position-dependence is realized in software by delegating interpretation of the signal to a procedure— a so-called *handler* or interpreter —which is local to the viewer in whose area the cursor momentarily appears. A surprising flexibility of command activation can be achieved in this manner by appropriate software. Various techniques have emerged in this connection, e.g. pop-up menus, pull-down-menus, etc. which are powerful even under the presence of a single button only. For many applications, a mouse with several keys is far superior, and the Oberon System basically assumes three buttons to be available. The assignment of different functions to the keys may of course easily lead to confusion when every application prescribes different key assignment. This is, however, easily avoided by the adherence to certain "global" conventions. In the Oberon System, the left button is primarily used for *marking* a position (setting a caret), the middle button for issuing general *commands* (see below), and the right button for *selecting* displayed objects.

Recently, it has become fashionable to use overlapping windows mirroring documents being piled up on one's desk. We have found this metaphor not entirely convincing. Partially hidden windows are typically brought to the top and made fully visible before any operation is applied to their contents. In contrast to the insignificant advantage stands the substantial effort necessary to implement this scheme. It is a good example of a case where the benefit of a complication is incommensurate with its cost. Therefore, we have chosen a solution that is much simpler to realize, yet has no genuine disadvantages compared to overlapping windows: tiled viewers as shown in Fig. 2.1.

2.2.2. Commands.

Position-dependent commands with fixed meaning (fixed for each type of viewer) must be supplemented by general commands. Conventionally, such commands are issued through the keyboard by typing the program's name that is to be executed into a special command text. In this respect, the Oberon System offers a novel and much more flexible solution which is presented in the following paragraphs. First of all we remark that a program in the common sense of a text compiled as a unit is mostly a far too large unit of action to serve as a command. Compare it, for example, with the insertion of a piece of text through a mouse command. In Oberon, the notion of a unit of action is separated from the notion of unit of compilation. The former is a command represented by a (exported) procedure, the latter is a module. Hence, a module may, and typically does, define several, even many commands. Such a (general) command may be invoked at any time by pointing at its name in any

text visible in any viewer on the display, and by clicking the middle mouse button. The command name has the form `M.P`, where `P` is the procedure's identifier and `M` that of the module in which `P` is declared. As a consequence, any command click may cause the loading of one or several modules, if `M` is not already present in main store. The next invocation of `M.P` occurs instantaneously, since `M` is already loaded. A further consequence is that modules are never

(automatically) removed, because a next command may well refer to the same module.

TODO: Fig. 2.1. Oberon display with tiled viewers

Every command has the purpose to alter the state of some operands. Typically, they are denoted by text following the command identification, and Oberon follows this convention. Strictly speaking, commands are denoted as parameterless procedures; but the system provides a way for the procedure to identify the text position of its origin, and hence to read and interpret the text following the command, i.e. the actual parameters. Both reading and interpretation must, however, be programmed explicitly.

The parameter text must refer to objects that exist before command execution starts and are quite likely the result of a previous command interpretation. In most operating systems, these objects are *files* registered in the directory, and they act as interfaces between commands. The Oberon System broadens this notion; the links between consecutive commands are not restricted to files, but can be any global variable, because modules do not disappear from storage after command termination, as mentioned above.

This tremendous flexibility seems to open Pandora's box, and indeed it does when misused. The reason is that global variables' states may completely determine and alter the effect of a command. The variables represent *hidden states*, hidden in the sense that the user is in general unaware of them and has no easy way to determine their value. The positive aspect of using global variables as interfaces between commands is that some of them may well be visible on the display. All viewers—and with them also their contents—are organized in a data structure that is rooted in a global variable (in module *Viewers*). Parts of this variable therefore constitute *visible states*, and it is highly appropriate to refer to them as command parameters.

One of the rules of what may be called the Oberon Programming Style is therefore to avoid hidden states, and to reduce the introduction of global variables. We do not, however, raise this rule to the rank of a dogma. There exist genuinely useful exceptions, even if the variables have no visible parts.

There remains the question of how to denote visible objects as command parameters. An obvious case is the use of the most recent selection as parameter. A procedure for locating that selection is provided by module Oberon. (It is restricted to text selections). Another possibility is the use of the caret position in a text. This is used in the case of inserting new text; the pressing of a key on the keyboard is also considered to be a command, and it causes the character's insertion at the caret position.

A special facility is introduced for designating viewers as operands: the star marker. It is placed at the cursor position when the keyboard's mark key (`SETUP`) is pressed. The procedure `Oberon.MarkedViewer` identifies the viewer in whose area the star lies. Commands which take it as their parameter are typically followed by an asterisk in the text. Whether the text contained in a text viewer, or a graph contained in a graphic viewer, or any other part of the marked viewer

is taken as the actual parameter depends on how the command procedure is programmed.

Finally, a most welcome property of the system should not remain unmentioned. It is a direct consequence of the persistent nature of global variables and becomes manifest when a command fails. Detected failures result in a trap. Such a trap should be regarded as an abnormal command termination. In the worst case, global data may be left in an inconsistent state, but they are not lost, and a next command can be initiated based on their current state. A trap opens a small viewer and lists the sequence of invoked procedures with their local variables and current values. This information helps a programmer to identify the cause of the trap.