Programming
Techniques and
Data Structures

Ellis Horowitz
Editor

Program Design by Informal English Descriptions

RUSSELL J. ABBOTT   The Aerospace Corp. and California State University

Russell J. Abbott is a Professor
of Computer Science at
California State University,
Northridge and a member of
the professional staff at the
Aerospace Corporation. His
current interests include
object-oriented and dataflow
approaches to software
architecture and the
application of knowledge
acquisition techniques to
requirements analysis.
Author's Present Address:
Russell J. Abbott, Computer
Science Department,
California State University—
Northridge, Northridge, CA
91330; The Aerospace
Corporation, M1-106,
P.O. Box 92957,
Los Angeles, CA 90009
ARPANET: Abbott @ Aero.

1. INTRODUCTION

This article discusses data types and their use in program design. A *data type* is a category of beings or things. Data types are useful for the logical organization of information and most people use them naturally in their everyday activities. Most people are not aware, however, that this familiar method of dividing the world into classes of objects can be directly related to the programming language notion of data types.

This article illustrates the use of data types in program design by demonstrating how an arbitrary procedure, initially stated in English, can be transformed into a program written in Ada.[1] The resulting program corresponds quite closely to the original English algorithm. Most striking is the parallel between the noun phrases in the original and the data types and program objects in the program. The bulk of this article is a discussion of that correspondence and of how to make the transformation between noun phrases and data types and objects.

We discuss program development from the traditional viewpoint of program design—where a program is a sequence of processing steps. We focus on the process by which one creates a program to achieve a programming goal, having been given an intuitive description of the goal. Most programmers find themselves in this situation most of the time. In solving such a problem, one must develop both a process to be performed and a collection of objects on which to perform it. Development of the process proceeds in parallel with the development of the object definitions. Although certainly not as formal as the methods of Dijkstra [3], Hoare [6], Wirth [10], and others, the focus is similar—the development of precise and understandable programs.

The method presented extends and makes more intuitive the approaches of, for example, Parnas [8] and Liskov [7]. In particular, the paper identifies common nouns as a natural

[1] Ada is a registered trademark of the Department of Defense.

**ABSTRACT: A technique is
presented for developing programs
from informal but precise English
descriptions. The technique shows
how to derive data types from
common nouns, variables from
direct references, operators from
verbs and attributes, and control
structures from their English
equivalents. The primary
contribution is the proposed
relationships between common
nouns and data types; the others
follow directly. Ada is used as the
target programming language
because it has useful program
design constructs.**

language analog to the programming language notion of abstract data types. This extends the current understanding of data types as follows. A type is currently understood as a collection of values and a collection of operations applicable to those values [4, 5]. Common nouns are more than that. A common noun names a class or concept even though there may be no values or operations yet defined for that class or concept. Common nouns provide a conceptual hook for a concept that may not be fully formed; they permit one to declare the existence of a concept whose details have yet to be worked out. In natural language, one frequently creates and uses a common noun before the details of the concept are thought through. Program design methodologies have not been so forgiving about data types, insisting that a data type is exactly the values and operations defined for them. As used in natural language, common nouns permit the need for a concept to be established before the concept itself is fully defined.

Associating common nouns with data types also makes the notion of data types more intuitive. Most people already have an intuitive understanding of common nouns. They will understand data types better when seen as analogous to common nouns.

This article is organized into 11 sections. Section 2 introduces the example through which the approach will be demonstrated. It also provides an overview of the approach. Section 3 discusses various classes of nouns and noun phrases. Section 4 derives the example problem data types. Section 5 derives the example problem data objects. Section 6 is a discussion of the difference between noun phrases that refer directly to objects and noun phrases that describe objects without guaranteeing a referent. Section 7 derives the operators of the example problem. Section 8 completes the transformation of the English statement of the problem solution into a program. Section 9 segments that solution into an problem specific part and a data type package. Section 10 applies the same method to another problem: the KWIC indexing problem. Section 11 presents some concluding thoughts.

2. AN APPROACH TO PROGRAM DEVELOPMENT
We present our technique by means of an example. Consider the following easy problem:

Write a function subprogram that, given two dates in the same year, returns the number of days between the two dates.

This is certainly a simple enough problem. The only difficulty is in specifying the calendar by which days are counted. Our current calendar is quite ad hoc. Its specification involves various rules and tables for dealing with special situations, e.g., leap year.

A good approach to this and similar situations is to create a calendar module (or **package** in Ada) in which calendar concepts are defined. Such a calendar package provides definitions for calendar data types and makes available operations on calendar objects of those types. Using those type and operation definitions, one then writes the desired program in terms of them. In this approach, the calendar concepts and operations are encapsulated within the calendar package, and only the data types and the operations on objects of those types are visible outside the package. Although this paper is not primarily about package construction methodology, it is about the use of packages for the definition of data types. Thus, a few brief words about packages may be appropriate.

(For a further discussion of data types and their use in programming, see [1–8, 10]).

A package encapsulating a data type or a related collection of data types permits the separation of two levels of programming concerns.

1. It permits one to separate
   (a) the details of the calendar that are important for any program dealing with calendar problems, that is, "how many days there are in February," from
   (b) the concern with the particular problem to be solved, that is, "how many days there are between two given days."
2. It also permit one to separate
   (a) the definitions of the data types and operators useful for solving a problem, that is, calendar data types and operators, from
   (b) considerations of how those data types and operators are actually implemented.

The use of a package containing the data types for a problem solution leads to a program of the following general structure:

```
package DATA_TYPE_PACKAGE is
    -- The package "visible part" in which the data types and
    -- operators on objects of those types declared. This part
    -- should also provide a conceptual model of the defined
    -- types. The conceptual model should provide enough in-
    -- formation so that the types and their operations can be
    -- used in other programs.
end DATA_TYPE_PACKAGE;

package body DATA_TYPE_PACKAGE is
    -- The package body in which the operators are imple-
    -- mented. The user of the package need never consult this
    -- part as long as he or she understands the conceptual
    -- model. Package bodies are not discussed further in this
    -- article.
end DATA_TYPE_PACKAGE;

with DATA_TYPE_PACKAGE   -- Provides access to the
                        -- package facilities.
procedure PROBLEM_SOLUTION is
    -- The particular problem is solved in terms of the declara-
    -- tions in the data type package.
end PROBLEM_SOLUTION;
```

This article shows how an analysis of the English statement of the problem and its solution can be used to guide the development of both the visible part of a useful package and the particular algorithm used for the given problem.

The approach focuses initially on the concrete, that is, on the solution to the given problem. Rather than first specifying the calendar package and then writing the program using capabilities from that package, we first write the program in terms of our intuitive understanding of the calendar and later formalize that understanding in the process of writing the program. Once finished with the program, we segregate the general calendar information from the problem-specific algorithm and offer it as a separate Ada calendar package. The resulting program and package will be similar to the one we might have developed had we approached it the other way, that is, by writing the package first; the difference is only in the way we get there. We take this approach not because we

believe that the specific problem solution is more important than the general package—in fact, we believe the opposite—but because this approach seems likely to be more comfortable for most programmers.

Our approach consists of three steps.

1. *Develop an informal strategy for the problem.* The informal strategy should state the problem solution on the same conceptual level as the problem itself. It should be expressed in problem domain terms.

2. *Formalize the informal strategy.* The second step is to formalize the solution by formalizing its data types, objects, operators, and control constructs.

3. *Segregate the solution into two parts: A package and a subprogram (or a collection of subprograms).* The package will contain the formalization of the problem domain, that is, the data types and their operators. The subprogram(s) will contain the specific steps (expressed in terms of the data types and operators defined in the package) for solving the particular problem.

2.1. An Informal Strategy

The first step is to develop an informal strategy for the problem. The informal strategy we shall use for the calendar problem is as follows:

1. If the two given dates are in the same month, the number of days between them is the difference between their days of the month.

2. If the two given dates are in different months, the following is done:

    (a) Determine the number of days from the earlier date to the end of its month. Keep track of that number in a counter called "Day_Counter."

    (b) For each month, starting with the first month after the earlier date and ending with the last month before the later date, add the number of days in that month to Day_Counter.

    (c) Add the day of the month of the later date to Day_Counter. Return that final sum as the number of days between the two dates.

To illustrate this solution, consider the two dates March 1 and August 18. The process would proceed as follows:

1. Since the two dates are not in the same month, the short solution does not apply.

2. Since the two dates are in different months, the following procedure must be used:

    (a) Determine that there are 30 days left in March after the earlier date. So Day_Counter is started off at 30.

    (b) The months between March and August are April, May, June, and July. Add 30 to Day_Counter for April, 31 for May, 30 for June, and 31 for July. At this point, Day_Counter has 152 days.

    (c) Add 18 to Day_Counter for the first part of August, giving a final result of 170 days between March 1 and August 18.

This informal approach expresses a solution to the problem in terms of the problem domain. It will serve as the informal strategy.

2.2 Formalizing the Strategy

Having developed an informal strategy, the next step is to formalize that strategy. The formalization steps are:

1. Identify the data types.

2. Identify the objects (program variables) of those types.

3. Identify the operators to be applied to those objects.

4. Organize the operators into the control structure suggested by the informal strategy.

We identify the data types, objects, operators, and control structures by looking at the English words and phrases in the informal strategy.

1. A common noun in the informal strategy suggests a data type.

2. A proper noun or direct reference suggests an object.

3. A verb, attribute, predicate, or descriptive expression suggests an operator.

4. The control structures are implied in a straightforward way by the English.

Although the process we follow in formalizing the strategy may appear mechanical, it is *not* (given the current state-of-the-art of computer science) an automatable procedure. The process of identifying the data types, objects, operators, and control structures, even given the English informal strategy, requires a great deal of real-world knowledge and an intuitive understanding of the problem domain. It is not just a matter of examining the English syntax.

2.3. A Preview of the Final Program

To eliminate any suspense (and as a guide to where we are headed), the approximate Ada program into which the informal strategy will be transformed by the end of this section follows. The subprogram implementations within the package body are not shown.

```
package CALENDAR_TOOLS is
   -- Conceptual Model
   -- This calendar package is based on the standard Grego-
   -- rian calendar. A slightly more detailed specification is
   -- presented later in the article. It is assumed that little
   -- explanation is required since the standard calendar is a
   -- familiar part of the everyday world. In general, a more
   -- complete conceptual model should be supplied when
   -- such universal understanding cannot be assumed.
   type DATE                           is private;
   type MONTH                          is ... ;
   type YEAR                           is ... ;
   type NUMBER_OF_DAYS    is new NATURAL;
   function ARE_IN_THE_SAME_MONTH(DATE_1,
                  DATE_2 : DATE) return BOOLEAN;
   function WHICH_DAY(DATE_I : DATE)
                           return NUMBER_OF_DAYS;
   function NUMBER_OF_DAYS_TO_END_OF_MONTH
                  (DATE_I : DATE) return NUMBER_OF_DAYS;
   function FIRST_MONTH_AFTER(DATE_I : DATE)
                           return MONTH;
   function LAST_MONTH_BEFORE(DATE_I : DATE)
                           return MONTH;
```

```
function NUMBER_OF_DAYS_IN_MONTH
        (MONTH_1 : MONTH, YEAR_1 : YEAR)
                        return NUMBER_OF_DAYS;
end CALENDAR_TOOLS;

with CALENDAR_TOOLS;     -- This makes the
use CALENDAR_TOOLS;      -- CALENDAR_TOOLS
                         -- capabilities available to
                         -- this program.
function NUMBER_OF_DAYS_BETWEEN(EARLIER DATE,
    LATER_DATE : DATE) return NUMBER_OF_DAYS is
DAY_COUNTER : NUMBER_OF_DAYS;
begin
  if ARE_IN_THE_SAME_MONTH
                (EARLIER_DATE, LATER_DATE) then
    return WHICH_DAY(LATER_DATE)
                       —WHICH_DAY(EARLIER_DATE)
  else  -- there is no need for a second condition since it is
        the inverse of the if condition            --
    DAY_COUNTER := NUMBER_OF_DAYS_TO_END_
                        OF_MONTH(EARLIER_DATE);
    for THAT_MONTH in
        FIRST_MONTH_AFTER(EARLIER DATE) ..
        LAST_MONTH_BEFORE(LATER_DATE) loop
        DAY_COUNTER := DAY_COUNTER +
                        NUMBER_OF_DAYS_IN_
                        MONTH(THAT_MONTH,
                        WHICH_YEAR(THAT_
                                        MONTH));
    end loop;
    DAY_COUNTER := DAY_COUNTER +
                        WHICH_DAY(LATER_DATE);
    return DAY_COUNTER;
  end if;
end NUMBER_OF_DAYS_BETWEEN;
```

Usually, most Ada programmers can immediately see how the English strategy would be transformed into this Ada program. The transformation is really quite simple and straightforward. The following sections provide a step-by-step discussion of the transformation process. While the discussion touches on many interesting ideas, the process becomes almost automatic for most programmers once the basic approach is understood.

It should be noted that this program may not work correctly if the months are given in the wrong order, that is, if the earlier month is given as the second parameter and the later month as the first. This point is pursued in more detail in the upcoming discussion on differentiating descriptive expressions from direct references.

3. NOUNS AND NOUN PHRASES
In formalizing the informal strategy, the first things to look at are the nouns and noun phrases in the informal strategy. Well-written Ada programs are usually object oriented. By "object-oriented" we mean that a primary focus of the program design is on the program objects. The phrase "object-oriented" originated with SMALLTALK76 and derived from programming languages such as SIMULA67. These systems emphasized the continued existence of objects that were in some sense independent of particular programs and could be manipulated by many programs. These objects were the forerunners of certain uses of Ada packages in that they had collections of operations defined on them that were independent of the programs that called the operations.

Object orientation emphasizes the importance of precisely identifying the objects and their properties to be manipulated by a program before starting to write the details of those manipulations. Without this careful identification, it is almost impossible to be precise about the operations to be performed and their intended effects.

The nouns and noun phrases in the informal strategy are good indicators of the objects and their classifications (i.e., data types) in our problem solution. Once one has a good grasp of the intuitive data types in the informal solution, it is much easier to develop the formal data types and objects to be used in the actual program. To aid in understanding this relationship, we digress from our original problem to discuss the various classes of nouns and noun phrases in English. (For a further discussion of words and their referents consult, e.g., [9].)

Nouns and noun phrases are used as references. We distinguish among three forms of reference:

Common nouns: A common noun (e.g., *apple*) is a name of a class of beings or things. For a term to be a common noun, it must be possible to use it in normal sentences with so called "limiting modifiers" such as *a, an, every, this, that, my,* etc.

Proper Nouns and Other Forms of Direct Reference: A proper noun (e.g., *The Big Apple*) is a name of a specific being or thing. A direct reference (e.g., *the apple used to poison Snow White*) is a reference to a specific, previously identified being or thing without necessarily referring to it by name— since it may not have a name.

Mass and Abstract Nouns and Units of Measure: A mass or abstract noun (e.g., *applesauce*) is a name of a quality, activity, or substance. We refer to this class of nouns simply as *mass nouns*. A unit of measure (e.g., *mouthful*) is a means of referring to a quantity of some quality, activity, or substance.

Some additional examples of nouns and their classifications follow.

Common Nouns: table, idea, image, rabbit, telephone, terminal, signal, filing cabinet, opera, pencil, job, message, book, computer, foot, launch, lip, beanbag

Proper Nouns and Direct References: Proper Nouns: Fortran, Niagara Falls, The Los Angeles Dodgers, Washington D.C., Europe; *Direct References:* my table, his rabbit, your error, this meatball, the tower of Pisa, the moon.

Mass Nouns and Units of Measure: Mass Nouns: oatmeal, traffic, cooking, software, equipment, heat, patriotism, furniture, documentation, energy, clothing, flexibility, programming, code, ice cream, usefullness, earth, air, fire, water, listening, music, information; *Units of Measure:* bowl (of oatmeal), box (of equipment), roomful (of furniture), scoop (of ice cream), lines (of code), meter (of poetry, length), second (of duration, arc), dyne (of force), number (of units), degree (of flexibility), measure (of music), acre (of land), acre–foot (of water), bit (of information).

3.1 Common Nouns vs Proper Nouns, Direct References, and Mass Nouns
Common nouns (e.g., *programming language*) are distinguished from proper nouns (e.g., *Ada*) and other forms of direct reference (e.g., *the DoD programming language*) in that common nouns do not refer to specific objects. Common

nouns refer to classes of objects, whereas proper nouns and direct references refer to specific, individual objects.

Common nouns (e.g., *lake*) are distinguished from mass nouns (e.g., *water*) and units of measure (e.g., *pint*) in that common nouns do not refer to qualities, activities, substances or their amounts. Common nouns refer to classes of individual objects; mass nouns refer to qualities (e.g., *maintainability*), activities (e.g., *work*), and substances (e.g., *salt*), that by themselves are not considered objects. Units of measure refer to quantities, sometimes determinate (e.g., *erg, gram*) and sometimes indeterminate (e.g., *man-month, pinch*), used for amounts of qualities, activities, and substances.

It is important to repeat the warning that differentiation among types of nouns is a matter of semantics and not a simple syntactic distinction. It generally requires knowledge of real-world phenomena and an understanding of the meaning of words before one can make the distinction properly.

The category that fits a particular term often depends on the use of the term and not just on the term itself. Note that the use of the term and not just on the term itself. Note that even the same word (e.g., *chicken*) may be used as both a common noun (Two chickens in every pot) and as a mass noun (I don't want any more chicken; it doesn't look very appetizing stuffed in the pot that way).

3.2 Direct References versus Descriptive Expressions
Direct references (e.g., *the DoD programming language*) must be distinguished not only from common nouns, but also from a similar but different form—descriptive expressions. Descriptive expressions (e.g., *the best programming language*) are not even necessarily referential. A direct reference is a reference to some specific object—one that is understood to have been previously identified. In contrast, a descriptive expression is only a description. There may or may not be an object that matches that description.

Before one can determine whether a descriptive expression actually has a referent, and, if so, what it is, a computation of some sort is required. The computation is often an attempt to find an object that matches the description along with an evaluation of the expression and an analysis of the object and its properties. Thus, while direct references are taken as indicative of objects, descriptive expressions are taken as indicative of operators—the operator that performs the computation. Section 6 includes some additional discussion of the distinction between direct references and descriptive expressions.

3.3. Mass Nouns and Their Units of Measure
Mass nouns are names of qualities, substances, and activities that do not have an *a priori* organization into individual units or instances. One never talks about "a water." Although one does refer to "this equipment" or "that sand," the reference is to a subdivision of the materials by some means other than by an *a priori* division into natural units. One refers to "a piece of furniture," suggesting that furniture is divided into natural pieces. But one also says "a piece of cheese" and "a piece of chocolate" when no natural unit is suggested. Even in the case of furniture (or equipment), the extent of the piece referred to is not usually knowable without further information, whereas "a rabbit" always refers to a single rabbit. "A piece of furniture" is more a reference to something that can be considered to be furniture than to a predeterminable unit of furniture. Thus, the referents of mass nouns are typically subdivided into arbitrary units.

A good test to determine whether a noun is a common or mass noun is to ask whether it can be used in the phrase,

"how much ———." The key word is "much." If the term is a mass noun, it fits in this phrase; if it is a common noun, one would substitute "many" for "much" and make the term plural.

> How much *chicken* do you want?
> versus
> How many *chickens* do you want?

3.4. Units of Measure
Names have been given to the arbitrary units used for subdividing mass noun referents. These units are called "units of measure" since they are used to quantify that which is not self-quantifying. The standard physical measurement systems (English, metric) define units of measure. Besides these traditional units of measure, other measurement systems are also common.

> *Wealth*: dollars, franks, pounds, marks, beads.
> *Beauty*: on a scale of 1 to 10.
> *Readability*: the fog index.
> *Computational complexity*: an $n \log(n)$ sort.
> *Quality*: a letter grade ('A', 'B', 'C,' 'D,' 'F,') or a class identification (USDA prime, choice, good, commercial, utility).
> *Reliability*: mean time until failure.
> *Funniness*: the reading on a laugh meter.
> *Intelligence*: IQ.

Our most basic unit of measure, however, is not a subdivision of a homogenous quantity but a simple integral count of the number of units of normal common nouns. When we answer the question "How many?", the answer is given as a count of the number of elements of a particular kind. Although in this case, the count is generally a count of objects that are referred to with a common noun (e.g., How many *apples?*), a count is still a measure of quantity and is considered a unit of measure.

One other class of nouns is frequently discussed—collective nouns. *Collective nouns* are nouns that refer to groups of individuals, where the group itself may be considered as an object. Examples of collective nouns include army, jury, audience, fleet, contents, crowd, group, mankind, remainder, and flock. Collective nouns may be treated as common nouns and need no special discussion here other than the following observation.

Care must be taken when transforming an informal strategy that uses a collective noun into a formal program. An operation performed on an object referred to by a collective noun (e.g., *sequester the jury*) generally affects the individual objects (i.e., the jurors) that make up the unit referred to by the collective noun. This is less a concern during the initial identification of the data types than it is at a latter stage in the program development process when the actual effects of operators are being specified.

Collective nouns often function as units of measure in much the same way as traditional units of measure are used for mass nouns. Thus a *herd* (of cows) is a unit of measure for the mass noun *cattle*. Other collective nouns also act as units of measure even though we may not have actual mass nouns for the equivalent concepts. Thus *herd* of elephants has the same sense as *herd* of cows, but we have no word to refer to elephants as an undifferentiated substance as we have for cattle.

4. THE DATA TYPES IN THE EXAMPLE PROBLEM
This section makes an initial identification of data types in our example problem by observing the common nouns and

units of measure in the informal strategy. In general, a common noun is taken as an indication of a data type, that is, either a private type or an enumeration type, and a unit of measure is taken as an indication of a type derived from a numeric type. The original, informal strategy with the common nouns and units of measures highlighted follows.

1. If the two given DATEs are in the same MONTH, the NUMBER_OF_DAYS between them is the difference between their DAYs of the MONTH.
2. If the two given DATES are in different MONTHS, the following is done.
   (a) Determine the NUMBER_OF_DAYS from the earlier DATE to the end of its MONTH. Keep track of that NUMBER in a COUNTER called "Day_Counter."
   (b) For each MONTH, starting with the first MONTH after the earlier DATE and ending with the last MONTH before the later DATE, add the NUMBER_OF_DAYS in that MONTH to Day_Counter.
   (c) Add the DAY of the MONTH of the later DATE to Day_Counter. Return that final SUM as the NUMBER_OF_DAYS between the two DATEs.

The highlighted terms are COUNTER, DATE, DAY, MONTH, NUMBER_OF_DAYS, and SUM. The first four of these are common nouns. Initially, they will be declared as data types but with no details supplied. Later they will be put into their own package. NUMBER_OF_DAYS and SUM are understood as synonymous. Since SUM is less informative than NUMBER_OF_DAYS, we will not use it as a data type. NUMBER_OF_DAYS is clearly a unit of measure and will be declared as a **type** derived from the **type** NATURAL. NATURAL is selected rather than a real type since NUMBER_OF_DAYS is understood to refer to a NATURAL rather than a real number.

These data types can serve as the initial collection of data types for the problem. We shall find later that one additional data type is needed and, that these data types are not all needed for this problem. As a first pass at declaring data types, we can write:

```
type COUNTER            is ... ;
type DATE               is ... ;
type DAY                is ... ;
type MONTH              is ... ;
type NUMBER_OF_DAYS     is new NATURAL;
```

Note that we did not include the term *number* as a separate common noun and hence, it was not declared as a data type. In some problems, number would properly be included as a data type. If we had a problem, for example, in which one had to consider different numbers and categorize them according to some property or other (e.g., prime numbers versus nonprime numbers), it would have been appropriate to consider *number* a data type. In this problem, however, *number* (or really *number of days*) is used as a unit of measure. The problem is asking: "How many" (number of) days.

It is not a crucial matter to get the data types exactly right initially. If too many or too few data types are declared at first, the ones that are not relevant can be eliminated and the necessary additional ones can be invented as the problem solution comes into better focus.

5. THE OBJECTS
Once the data types are determined (at least to a first approximation), the next step is to identify the program objects. The

objects are found by looking at the proper nouns and the direct references in the informal description. Recall from the earlier discussion that a proper noun is a name of a particular being or thing and a direct reference is a reference to a particular being or thing but without giving it a name. (Recall also that there is a distinction between direct references and descriptive expressions: a direct reference refers to a known object, whereas a descriptive expression provides a description for which no object is necessarily known to exist. However, see the more detailed discussion in Sect. 6 contrasting direct references with descriptive expressions.)

The proper nouns and the direct references are references to the presumed objects that the informal description manipulates to get its answer. The original informal description with the proper nouns and direct references highlighted follows.

1. If THE_TWO_GIVEN_DATES are in the same month, the number of days between them is the difference between their days of the month.
2. If THE_TWO_GIVEN_DATES are in different months, the following is done.
   (a) Determine the number of days from THE_EARLIER_DATE to the end of its month. Keep track of THAT_NUMBER in a counter called "DAY_COUNTER."
   (b) For each month, starting with the first month after THE_EARLIER_DATE and ending with the last month before THE_LATER_DATE, add the number of days in THAT_MONTH to DAY_COUNTER.
   (c) Add the day of the month of THE_LATER_DATE to DAY_COUNTER, and return THAT_FINAL_SUM as the number of days between THE_TWO_DATES.

The highlighted terms suggest that the objects are:

THE_EARLIER_DATE,
THE_LATER_DATE,
  THAT_NUMBER   -- the number of days from the first
                -- date to the end of its month,
  DAY_COUNTER   -- the counter,
  THAT_MONTH   -- a month reference that varies over
                -- the months ranging between the
                   two dates, and
THE_FINAL_SUM.

We can now extend our original set of declarations by adding declarations for these objects.

```
type COUNTER              is ... ;
type DATE                 is ... ;
type DAY                  is ... ;
type MONTH                is ... ;
type NUMBER_OF_DAYS       is new NATURAL;
THE_EARLIER_DATE          : DATE;
THE_LATER_DATE            : DATE;
THAT_NUMBER_OF_DAYS       : NUMBER_OF_DAYS
DAY_COUNTER               : COUNTER;
THAT_MONTH                : MONTH;
THE_FINAL_SUM             : NUMBER_OF_DAYS;
```

Again, these objects are a first approximation to the final program variables. As the program comes into better focus, the collection of objects will become clearer.

6. DIRECT REFERENCE AND DESCRIPTIVE EXPRESSIONS

It is not easy to distinguish between direct references, which are associated with objects, and descriptive expressions, which are associated with operators. Note, for example, that we consider THE_EARLIER_DATE to be a direct reference (and hence to be associated with an object), but that in the next section we consider the term "the first month after (the earlier date)" to be a descriptive expression (and hence to be associated with an operator).

Determining in which of the two categories a term belongs is difficult. The choice depends upon whether the term in question is used as a name or whether it implies some computation. We make the following rules.

If a term refers to something that is already known to exist, and the term is simply a way to refer to that known thing, then it is considered a direct reference and is associated with an object.

In this example, the term "the earlier date" identifies one of two known objects (the dates that are already given), and the reference is simply a way of identifying one of them—since no names had been given to either.

If the term describes a possible object whose identity (and possibly even whose existence) must be determined by some computation—no matter how simple the computation—then the term is a descriptive expression and is associated with the operator that performs that computation.

In this example, the term "the first month after (the earlier date)" implies a computation (probably either a search or lookup) to determine the month that is the first one after the given month.

This is a difficult distinction to make and in making it, one must rely on one's understanding of the problem. In our example of THE_EARLIER_DATE, one could argue that computation is required to tell which of the two given dates is the earlier. In this case, however, the implication in the problem (or at least the assumption we have been making) is that the two dates are given in a prescribed order—the earlier one first, for example—so that no computation is required. The reference to the earlier date is treated as a name for that date rather than as a description that provides a means for selecting one (the earlier) of the two given dates.

On the other hand, the problem might have been understood differently. Perhaps the two dates are not given in a specific order. In that case, "the earlier date" would not be a direct reference but would, indeed, be a descriptive expression associated with an operator. The operator with which it would be associated would be the one that, given two dates, determines which of the two is the earlier.

An added minor complexity would have to be faced in this alternate interpretation. What if the two dates are the same? Then, neither one is earlier and "earlier" would have to be understood to mean "earlier or select one arbitrarily if the two are the same." Under our original interpretation, THE_EARLIER_DATE is understood to refer to the date given in the earlier position, so even if the two dates were the same (or, in fact, even if the "earlier" were chronologically later than the "later"), there would be no need to reinterpret the meaning of "earlier." (Of course, in the actual problem, this concern is needless anyway. If the two dates were the same, they would

be in the same month, and one would never refer to "the earlier" date at all: the algorithm would not get to that point.)

7. THE OPERATORS

In making an initial identification of the operators, we look at the verbs, attributes, predicates, and descriptive expressions in the informal strategy.

Verb: A verb expresses an act or occurrence. These are usually represented by procedure subprograms that take as arguments the objects on which the action is to be performed, and that perform the action on those objects.

Attribute: An attribute is a property, characteristic, association, or component of something. These are usually represented by function subprograms that take as arguments the object whose attribute is desired, and that return as their values the value of the attribute for that object.

Predicate: A predicate designates a property or relation that can be considered TRUE or FALSE, that is, to hold or not to hold. These are usually represented as function subprograms that take as their arguments the objects about which it is desired to know if the predicate holds, and that return as their values either TRUE or FALSE.

Descriptive Expression: A descriptive expression is a (presumably definitive) characterization for which there may be some particular object(s). These are usually represented as function subprograms that take as their arguments certain characteristics of the desired object, and that return as their values the object fitting the description.

The same informal description follows again, only this time with the verbs, attributes, predicates, and descriptive expressions highlighted.

1. If the two given dates ARE_IN_THE_SAME_MONTH, THE_NUMBER_OF_DAYS_BETWEEN them is the DIF-FERENCE_BETWEEN their DAYS_OF_THE_MONTH.

2. If the two given dates ARE_IN_DIFFERENT_MONTHS, the following is done:
   (a) Determine the NUMBER_OF_DAYS_FROM_the earlier date_TO_THE_END_OF_ITS_MONTH. KEEP_TRACK_OF_that number_IN a counter called "Day_Counter."
   (b) For each month, starting with the FIRST_MONTH_AFTER the earlier date and ending with the LAST_MONTH_BEFORE the later date, ADD_the NUM-BER_OF_DAYS_IN that month_TO Day_Counter.
   (c) ADD_the DAY_OF_THE_MONTH_OF the later date_TO Day_Counter. RETURN that final sum as the NUMBER_OF_DAYS_BETWEEN the two dates.

The highlighted words suggest that the operators are:

ARE_IN_THE_SAME_MONTH ⟨date_1⟩, ⟨date_2⟩;
NUMBER_OF_DAYS_BETWEEN ⟨date_1⟩, ⟨date_2⟩;
DIFFERENCE_BETWEEN ⟨day_1⟩, ⟨day_2⟩;
DAY_OF_THE_MONTH ⟨date⟩;
ARE_IN_DIFFERENT_MONTHS ⟨date_1⟩, ⟨date_2⟩;
NUMBER_OF_DAYS_FROM_⟨date⟩_TO_THE_END_OF_ MONTH;
KEEP_TRACK_OF_⟨number_of_days⟩_IN ⟨counter⟩;
FIRST_MONTH_AFTER ⟨date⟩;
LAST_MONTH_BEFORE ⟨date⟩;
NUMBER_OF_DAYS_IN ⟨month⟩;

ADD_⟨number_of_days⟩_TO_⟨counter⟩;
RETURN ⟨number_of_days⟩;

It is more useful to display these in the form of Ada subprogram signatures. In that form, the formal parameters and their data types are shown. That form also indicates whether the subprogram is a **procedure** or a **function**, and if a **function**, the data type of the returned value.

function ARE_IN_THE_SAME_MONTH(DATE_1,
                      DATE_2 : DATE) **return** BOOLEAN;
function NUMBER_OF_DAYS_BETWEEN(DATE_1,
                    DATE_2 : DATE) **return** NUMBER_OF_DAYS;
  -- This is the function being defined.
function DIFFERENCE_BETWEEN
        (DAY_I, DAY_J : NUMBER_OF_DAYS)
                   return NUMBER_OF_DAYS;
  -- The two days are guaranteed to be in the same month
  -- when this function is used.
function WHICH_DAY(DATE_I : DATE)
                   return NUMBER_OF_DAYS;
  -- The DAY_OF_THE_MONTH function. We understand
  -- that the value returned is a NUMBER_OF_DAYS into
  -- the month rather than an "identity" of a particular day
  -- of the month. If the value returned were a particular,
  -- identified DAY (e.g., the "ides" of the month), the type
  -- of the value returned would be a DAY rather than a
  -- NUMBER_OF_DAYS.

function ARE_IN_DIFFERENT_MONTHS
        (DATE_1, DATE_2 : DATE) **return** BOOLEAN;
  -- This function is the negation of the first one and will
  -- not be used.
function NUMBER_OF_DAYS_TO_END_OF_MONTH
        (DATE_I : DATE) **return** NUMBER_OF_DAYS;
procedure KEEP_TRACK_OF_IN
        (DAYS_I : NUMBER_OF_DAYS;
                COUNTER_I : **out** COUNTER);
  -- The COUNTER will be Day_Counter.
function FIRST_MONTH_AFTER (DATE_I : DATE)
                  return MONTH;
function LAST_MONTH_BEFORE (DATE_I : DATE)
                  return MONTH;
function NUMBER_OF_DAYS_IN_MONTH
        (MONTH_I : MONTH, YEAR_I : YEAR)
                 return NUMBER_OF_DAYS;
  -- The YEAR parameter is necessary because it is not
  -- possible to tell the number of days in a month from the
  -- month alone.
procedure ADD_TO (DAYS_I : NUMBER OF DAYS;
               COUNTER_I : **in out** COUNTER);
  -- Add the two values together and leave the sum in
  -- COUNTER_I.
  -- We do not define a "return" operator because we know
  -- that an analogous feature is built into Ada as a primi-
  -- tive.

Inspection of the list of data types and operators supports our initial proposal to derive NUMBER_OF_DAYS as a **new type** from NATURAL. We can then identify the **type** COUNTER with NUMBER_OF_DAYS. For the DIFFERENCE operator we can use "−", the INTEGER subtract operator; for KEEP_TRACK_OF we can use " :=", the assignment operator; for ADD_TO we can use a combination of the INTEGER add operator "+" and assignment.

8. THE INFORMAL STRATEGY REWRITTEN
The original informal strategy, fully transformed with respect to data types, objects, and operators follows. Only the control structure remains from the original.

function NUMBER_OF_DAYS_BETWEEN
        (EARLIER_DATE, LATER_DATE : DATE)
               return NUMBER_of_DAYS is
type DATE   is . . . ;
type DAY   is . . . ;
type MONTH is . . . ;
type YEAR   is . . . ; -- necessary for the operator
                    -- NUMBER_OF_DAYS_IN_MONTH
type NUMBER_OF_DAYS is new NATURAL;

THAT_NUMBER_OF_DAYS	: NUMBER_OF_DAYS
DAY_COUNTER	: NUMBER_OF_DAYS;
THAT_MONTH	: MONTH;
THE_FINAL_SUM	: NUMBER_OF_DAYS;

function ARE_IN_THE_SAME_MONTH
        (DATE_1, DATE_2 : DATE)
                 return BOOLEAN;
function WHICH_DAY (DATE_I : DATE)
                 return NUMBER_OF_DAYS;
function NUMBER_OF_DAYS_TO_END_OF_MONTH
        (DATE_I : DATE) **return** NUMBER_OF_DAYS;
function FIRST_MONTH_AFTER (DATE_I : DATE)
                 return MONTH;
function LAST_MONTH_BEFORE (DATE_I : DATE)
                 return MONTH;
function NUMBER_OF_DAYS_IN_MONTH
        (MONTH_I : MONTH, YEAR_I : YEAR)
                 return NUMBER_OF_DAYS;
  -- Note that the operators on NUMBER_OF_DAYS are
  -- omitted. Instead we use the standard INTEGER operators.

1. If ARE_IN_THE_SAME_MONTH (EARLIER_DATE, LATER_DATE), the number of days between them is WHICH_DAY (LATER_DATE)−WHICH_DAY (EARLIER_DATE).

2. If **not** ARE_IN_THE_SAME_MONTH (EARLIER_DATE, LATER_DATE), the following is done.

  (a) DAY_COUNTER :=
     NUMBER_OF_DAYS_TO_END_OF_MONTH
     (EARLIER_DATE).

  (b) For each month, called THAT_MONTH, starting with FIRST_MONTH_AFTER (EARLIER_DATE) and ending with LAST_MONTH_BEFORE (LATER_DATE),

  (c) DAY_COUNTER := DAY_COUNTER + NUMBER_OF_DAYS_IN_MONTH (THAT_MONTH, this_year).

  (d) We do not yet know how to determine what this_year is.

  (e) DAY_COUNTER := DAY_COUNTER + WHICH_DAY (LATER_DATE).
    return DAY_COUNTER.

Notice that there is a data type in the restated program that was not found in the original informal description. The function NUMBER_OF_DAYS_IN_MONTH has two arguments: a MONTH and a YEAR. The YEAR parameter is necessary

because it is not possible to tell, in general, how many DAYS there are in a MONTH without knowing the YEAR. We are thus forced to use an additional data type YEAR. This should not be disturbing; our initial declarations should not be expected to be perfect. Also, we need an operator that returns the current YEAR, given a date.

function WHICH_YEAR (DATE_1 : DATE) **return** YEAR;

All that remains now is to translate the informal control structure to a formal control structure. There are two obvious control constructs in the informal description: an **if-then-else** construct and a **for loop** construct. There is no question about how to translate the **if-then-else** construct.

The one issue to be resolved about the **for** construct is the range of its loop parameter. Inspection of the English version of the **for** statement (and our knowledge of the calendar) suggests that **type** month should be declared as an enumeration type so that the **for** statement parameter can be declared as a subrange of that type. If we declare

type MONTH is (JANUARY,
FEBRUARY, . . . , DECEMBER);

we can then let the discrete range for the **for** parameter be

MONTH'SUCC(WHICH MONTH (EARLIER_DATE)) . .
MONTH'PRED(WHICH_MONTH(LATER_DATE)).

Unfortunately, this expression is not completely defined; we have not yet defined a WHICH_MONTH operator. Our original strategy was to do in one step what we now are considering doing in two, that is, we originally used:

FIRST_MONTH_AFTER (DATE_1) for
MONTH'SUCC(WHICH_MONTH (DATE_1))
and
LAST_MONTH_BEFORE(DATE_1) for
MONTH'PRED(WHICH_MONTH (DATE_1))

The revised version is much cleaner, if, unfortunately, somewhat longer. It makes the two steps explicit:

1. Determine which month a given date is in.
2. Determine another month (successor or predecessor) from a given month.

It uses both a new **function** WHICH_MONTH, that returns the month of a given date, and the attributes SUCC and PRED, that are built into all Ada enumeration types.
We therefore need:

function WHICH_MONTH(DATE_1 : DATE) **return** MONTH;

Finally, note that the revised version of the program does not use two of our original data types: DAYS and COUNTER. What has happened is that both DAYS and COUNTER turned out to be the same data type as NUMBER_OF_DAYS and we simply selected NUMBER_OF_DAYS as the name of that data type.

Actually what we did was slightly trickier than that. The data type COUNTER is not really the same thing as the data type NUMBER_OF_DAYS. A COUNTER is an object that has the ability to count, and in this particular example, it has the ability to count values of the type NUMBER_OF_DAYS. A COUNTER itself, however, is not a NUMBER_OF_DAYS. In this example, though, there was no need to consider a COUNTER object as an independent thing distinct from the value that it counted. So we were able to identify the COUNTER with its counted value NUMBER_OF_DAYS without

loss. There may be cases in other problems in which the existence of a COUNTER itself is important independent of the value that it is counting. In those cases, the data type COUNTER and the data type of the counted value must be kept separate.

Note that the program does not use the objects THAT_NUMBER_OF_DAYS and THE_FINAL_SUM. It also does not use the object THAT_MONTH except within the **for** loop, so there is no need to declare it initially.

With these modifications, the following, finally, is the original informal strategy fully transformed into an Ada program. The subprograms, of course, are not yet implemented.

```
function NUMBER_OF_DAYS_BETWEEN
            (EARLIER_DATE, LATER_DATE : DATE)
                          return NUMBER_OF_DAYS is

   type DATE    is . . . ;
   type MONTH is . . . ;
   type YEAR    is . . . ;   -- necessary for the operator
                             -- NUMBER_OF_DAYS_IN_
                             -- MONTH
   type NUMBER_OF_DAYS is new NATURAL;
   DAY_COUNTER : NUMBER_OF_DAYS;
   function ARE_IN_THE_SAME_MONTH
            (DATE_1, DATE _2 : DATE) return BOOLEAN;
   function WHICH_DAY (DATE_1 : DATE)
                          return NUMBER_OF_DAYS;
   function NUMBER_OF_DAYS_TO_END_OF_MONTH
            (DATE_1 : DATE) return NUMBER_OF_DAYS;
   function WHICH_MONTH (DATE_1 : DATE)
                          return MONTH;
   function NUMBER_OF_DAYS_IN_MONTH
            (MONTH_1 : MONTH, YEAR_1 : YEAR)
                          return NUMBER_OF_DAYS;
   -- Note that the operators on NUMBER_OF_DAYS are
   -- omitted. Instead we use the standard INTEGER opera-
      tors.
begin
   if ARE_IN_THE_SAME_MONTH
                (EARLIER_DATE, LATER_DATE) then
        return WHICH_DAY(LATER_DATE)
                          - WHICH_DAY(EARLIER_DATE)
   else -- there is no need for a second condition
        -- since it is the inverse of the if condition
        DAY_COUNTER :=
          NUMBER_OF_DAYS_TO_END_OF_MONTH
                          (EARLIER_DATE);
        for THAT_MONTH in
          MONTH'SUCC(WHICH_MONTH
                          (EARLIER_DATE)) . .
          MONTH'PRED(WHICH_MONTH
                          (LATER_DATE)) loop
     DAY_COUNTER := DAY_COUNTER +
        NUMBER_OF_DAYS_IN_MONTH (THAT_MONTH,
                          WHICH_YEAR(THAT_MONTH));
        end loop;
        DAY_COUNTER := DAY_COUNTER +
                          WHICH_DAY (LATER_DATE);
        return DAY_COUNTER;
   end if;
end NUMBER_OF_DAYS_BETWEEN;
```

9. SPECIFYING A PACKAGE FOR THE PROBLEM
The function displayed above solves the problem as given.

The disadvantage of leaving the solution as a subprogram is that the calendar data types and operators defined internally are not available to other programs outside of this one. So the next step is to extract the calendar data type declarations and operators from the subprogram body and to assemble them in a separate package. They would then be available for use by any program that needs them. Thus, the next job is to develop a package specification upon which the solution could be built.

In specifying a **package**, we provide:

1. The motivation for the **package**, that is, the **package** requirements.
2. A conceptual model for the **package**. A package conceptual model is a description of what a user of the package should understand about it in order to make use of it. Since one of the major features of a package is that its users need not understand how the facilities it offers are implemented, a package conceptual model is not a description of the package implementation. The other side of that coin, however, is that since the users do not know how the package is implemented, they must be given something to hang onto so that they can make any use of it at all. So the conceptual model is the presentation to the user of how the package should be understood.

    As an example, consider a package encapsulating a stack and another package encapsulating a queue. Conceptual models for these might present pictures showing a "tube" in which objects are entered at one end and removed either from the same end for a stack or from the other end for a queue. Of course, if the operators are specified formally enough, the conceptual model is redundant. But most people find it easier to understand a model than a pure formalism, and the conceptual model is intended to provide just such an intuitive device.
3. Specifications for the operators that the **package** makes available. We provide those specifications in terms of the pre and post conditions that define those operators.

This paper is not about package development strategies. Were that its focus, this discussion of package structure and organization would be greatly expanded. (For more discussion about package development strategies, any of the available books and articles on Ada should be consulted, [1, 2].)

We declare a **package** for the calendar types by removing the nonproblem specific data type and operator declarations from the problem solution. The nonproblem specific data types are those that seem likely to have a broader use than for this particular problem alone. This is, of course, a matter of judgment.

```
package CALENDAR_TOOLS is
    -- Requirements
    -- Data types and operators for
    -- working with calendar information.
    -- Conceptual Model
    -- The standard Gregorian calendar.
    type NUMBER_OF_DAYS is new NATURAL;
    type MONTH is (JANUARY, FEBRUARY, MARCH,
                   APRIL, MAY, JUNE, JULY, AUGUST,
                   SEPTEMBER, OCTOBER, NOVEMBER,
                   DECEMBER);
    type YEAR is NATURAL range <some range>;
    type DATE is private;   -- presumably including refer-
                            -- ences to (Year, Month, Day)
    -- Invariant
    -- Every object of type DATE must conform to
```

```
    -- the Gregorian calendar: no month may
    -- have an illegal date.
    -- Operators
    function IS_LEAP_YEAR(Y : YEAR) return BOOLEAN;
    -- Returns TRUE if
    -- either  Y mod 400 = 0    (All years divisible by 4
    --                           are leap years except for
    -- or                       those century years that
    --          Y mod 4 = 0.    are not divisible by 400.)
    -- and
    --          Y mod 100 /= 0
    -- (Note: this either/or construct is not an Ada
    -- construct but is used for specification purposes.)
    function WHICH_DAY (DATE_I : DATE)
                            return NUMBER_OF_DAYS;
    -- If DATE_I = (Y_I, M_I, D_I), returns D_I.
    function WHICH_MONTH (DATE_I : DATE)
                            return MONTH;
    -- If DATE_I = (Y_I, M_I, D_I), returns M_I.
    function WHICH_YEAR (DATE_I : DATE) return YEAR;
    -- If DATE_I = (YEAR_I, MONTH_I, DAY_I), returns
    -- YEAR_I.
    function NEW_DATE (YEAR_I, MONTH_I, DAY_I)
                            return DATE;
    -- If (YEAR_I, MONTH_I, DAY_I) is a valid Gregorian
    -- date, returns as a value the DATE (YEAR_I,
    -- MONTH_I, DAY_I).
    function NUMBER_OF_DAYS_IN_MONTH
            (MONTH_I: MONTH; YEAR_I: YEAR)
                            return NUMBER_OF_DAYS;
    -- Returns the number of days in MONTH_I in
    -- YEAR_I. The results would differ for the same
    -- month in different years only if the month were
    -- February and year were leap year or not.
end CALENDAR_TOOLS;
```

Notice that only a few of the needed operators are actually listed in the **package** specification. These are the primitive operators. The others, such as NUMBER_OF_DAYS_TO_END_OF_MONTH, can be defined in terms of the primitive operators.

The final program is:

```
with CALENDAR_TOOLS;
use CALENDAR_TOOLS;
function NUMBER_OF_DAYS_BETWEEN
        (EARLIER DATE, LATER_DATE : DATE)
                            return NUMBER_OF_DAYS is
function NUMBER_OF_DAYS_TO_END_OF_MONTH
        (DATE_I : DATE) return NUMBER_OF_DAYS is
begin
    return    NUMBER_OF_DAYS_IN_MONTH (WHICH_
              MONTH (DATE_I), WHICH_YEAR (DATE_I))
              WHICH_DAY (DATE_I);
end NUMBER_OF_DAYS_TO_END_OF_MONTH;
begin
    if WHICH_MONTH(EARLIER_DATE)
              = WHICH_MONTH(LATER_DATE) then
        return WHICH_DAY(LATER_DATE)
                  - WHICH_DAY(EARLIER_DATE)
    else  -- There is no need for a second condition since it
          -- is the inverse of the if condition.
        DAY_COUNTER :=
              NUMBER_OF_DAYS_TO_END_OF_MONTH
                            (EARLIER_DATE);
```

```
for THAT_MONTH in
      MONTH'SUCC(WHICH_MONTH
                          (EARLIER DATE)) ..
      MONTH'PRED(WHICH_MONTH
                          (LATER_DATE)) loop
DAY_COUNTER := DAY COUNTER +
      NUMBER_OF_DAYS_IN_MONTH(THAT_MONTH,
                  WHICH_YEAR(THAT_MONTH));
   end loop;
   DAY_COUNTER := DAY_COUNTER +
                  WHICH_DAY(LATER_DATE);
      return DAY_COUNTER;
   end if;
end NUMBER_OF_DAYS_BETWEEN;
```

10. ANOTHER QUICK EXAMPLE

As another example of the approach, we design the top level of a KWIC indexing program. A KWIC index of titles is an alphabetized list where each title is included in the list once for each of its keywords. A word is considered a keyword if it is not an article, preposition, or other overly common word.

For example, if the input titles were

Abstract Data Types and Software Validation
On the Criteria to be Used in Decomposing Systems into Modules
On the Design and Development of Program Families
A Design Methodology for Reliable Software Systems

the KWIC index would be

Abstract Data Types and Software Validation
On the **Criteria** to be Used in Decomposing Systems into Modules
Abstract **Data** Types and Software Validation
On the Criteria to be Used in **Decomposing** Systems into Modules
On the **Design** and Development of Program Families
On the Design and **Development** of Program Families
On the Design and Development of Program **Families**
A **Design** Methodology for Reliable Software Systems
A Design **Methodology** for Reliable Software Systems
On the Criteria to be Used in Decomposing Systems into **Modules**
On the Design and Development of **Program** Families
A Design Methodology for **Reliable** Software Systems
A Design Methodology for Reliable **Software** Systems
Abstract Data Types and **Software** Validation
A Design Methodology for Reliable Software **Systems**
On the Criteria to be Used in Decomposing **Systems** into Modules
Abstract Data Types and Software **Validation**

Assume a program that takes a list of titles and produces a KWIC index. An informal solution is:

1. Consider each title in the title list in turn.
2. For each title, consider each word in that title.
3. If the word is not a common word, enter the title into the KWIC index at the position identified by that and perhaps subsequent words.

10.1. The Data Types

The common nouns are:

TITLE,
TITLE_LIST,
WORD,

COMMON_WORD,
KWIC_INDEX,
POSITION,
THAT_AND_SUBSEQUENT_WORDS

Notice that we have listed THAT_AND_SUBSEQUENT_WORDS as a data type. This is certainly not one of the more obvious types and it was discovered only after a fair amount of analysis. This is an "ugly" data type, but it does characterize, on the level of the problem, the information required to determine where a title fits in a KWIC Index with respect to a given word in the title. We can define the initial data types as:

type TITLE	is ...;
type TITLE_LIST	is ...;
type WORD	is ...;
type COMMON_WORD	is ...;
type KWIC_INDEX	is ...;
type POSITION	is ...;
type THAT_AND_SUBSEQUENT_WORDS	is ...;

10.2. The Objects

The objects are:

THE_TITLE_LIST;
EACH_TITLE, THAT_TITLE, THE_TITLE; (These are used synonymously.)
EACH_WORD, THE_WORD, THE_CURRENT_WORD; (These are used synonymously.)
THE_KWIC_INDEX;
THE_POSITION.

So we can make our initial object declarations:

THE_TITLE_LIST	:	TITLE_LIST;
THE_TITLE	:	TITLE;
THE_WORD	:	WORD;
THE_KWIC_INDEX	:	KWIC_INDEX;
THE_POSITION	:	POSITION;

10.3. The Operators

The operators (in the form of Ada subprogram signatures) are:

```
procedure CONSIDER_NEXT_TITLE
      (NEXT_TITLE   :    out TITLE;
       A TITLE_LIST : in out TITLE_LIST;
       MORE_TITLES :    out BOOLEAN);
--  Given a TITLE_LIST, sets NEXT_TITLE to the next
--  unscanned TITLE and increments the TITLE_LIST's in-
--  ternal "scan pointer." Sets MORE_TITLES depending
--  on whether or not an unscanned title has been found
--  in the LIST.
procedure CONSIDER_FIRST_TITLE
      (FIRST_TITLE   :    out TITLE;
       A_TITLE_LIST : in out TITLE_LIST;
       ANY_TITLES   :    out BOOLEAN);
--  Good practice suggests an initialization "first" operator
--  to complement the "next" operator. The meanings of
--  the parameters are similar.
procedure CONSIDER_NEXT_WORD
      (NEXT_WORD   :    out WORD;
       A_TITLE         : in out TITLE;
       MORE_WORDS :    out BOOLEAN);
--  Similar to the CONSIDER_NEXT_TITLE operator ex-
--  cept that the object scanned is a single TITLE and the
--  object scanned for is a WORD. In addition, titles have a
--  "current word" indicator that is set to the NEXT_
--  WORD.
```

procedure CONSIDER_FIRST_WORD
  (FIRST_WORD :   **out** WORD;
  A_TITLE   : **in out** TITLE;
  ANY_WORDS :   **out** BOOLEAN);
-- A "first" WORD operator to complement the "next"
-- WORD operator.
function IS_A_COMMON_WORD (A_WORD : WORD)
                    return BOOLEAN;
-- Determines whether the given WORD is a
-- COMMON_WORD.
procedure ENTER(A_TITLE   : TITLE;
            A_KWIC_INDEX : KWIC_INDEX;
            A_POSITION   : POSITION);
-- Enters the TITLE into the KWIC_INDEX at the indi-
-- cated position.
function REMAINING_WORDS(A_TITLE : TITLE)
      return THAT_AND_SUBSEQUENT_WORDS;
-- Returns the sequence of words remaining in the title
-- from the "current word" (see CONSIDER_NEXT_
-- WORD) to the end of the title. In Ada, the name of a
-- subprogram may not be the same as the name of a data
-- type.
function WHICH_POSITION
  (WORDS   : THAT_AND_
               SUBSEQUENT_WORDS;
  A_KWIC_INDEX : KWIC_INDEX) **return** POSITION;

-- Finds the POSITION in the KWIC_INDEX where titles
-- are indexed under the "current and perhaps subse-
-- quent word(s)" in the given TITLE.

10.4 The Control Structure and the Final Solution
The control structure is a simple pair of nested loops. They are shown clearly in the final program.

The final program consists of two **packages** and a **proce-dure** that uses those **packages**. One **package** defines the data types and the operators for WORDs and TITLEs and the other defines the data types and operators for KWIC objects.

package TITLES_AND_WORDS **is**
-- **Requirements**
-- Data types for dealing with titles that consist of se-
-- quences of words.
-- **Conceptual Model**
-- The cleanest way to give the conceptual model is to
-- make use of a previously defined, **generic package** for
-- general sequences. Since generics are not discussed
-- here, we do not give the complete specification. We do
-- assume, though, that title sequences have pointers that
-- may be set and moved.

type THAT_AND_SUBSEQUENT_WORDS **is private;**
type TITLE   **is private;**
type TITLE_LIST   **is private;**
type WORD   **is private;**
-- **Operators**
-- We do not repeat the descriptions of the operators.
-- They are the same as given earlier.
procedure CONSIDER_FIRST_TITLE
  (FIRST_TITLE :   **out** TITLE;
  A_TITLE_LIST : **in out** TITLE_LIST;
  ANY_TITLES :   **out** BOOLEAN);
procedure CONSIDER_NEXT_TITLE
  (NEXT_TITLE :   **out** TITLE;

A_TITLE_LIST : **in out** TITLE_LIST;
MORE_TITLES :   **out** BOOLEAN);
procedure CONSIDER_FIRST_WORD
  (FIRST_WORD :   **out** WORD;
  A_TITLE   : **in out** TITLE;
  ANY_WORDS :   **out** BOOLEAN);
procedure CONSIDER_NEXT_WORD
  (NEXT_WORD :   **out** WORD;
  A_TITLE   : **in out** TITLE;
  MORE_WORDS :   **out** BOOLEAN);
function REMAINING_WORDS(A_TITLE : TITLE)
        return THAT_AND_SUBSEQUENT_WORDS;
end TITLES_AND_WORDS;
with TITLES_AND_WORDS;
use TITLES_AND_WORDS;
package KWIC_STUFF **is**
-- **Requirements**
-- Data types for KWIC indices.
-- **Conceptual Model**
-- Again, no complete conceptual model is given since it
-- too is best described in terms of a **generic** sequence
-- **package**.
type COMMON_WORD **is private;**
type KWIC_INDEX   **is private;**
type POSITION   **is private;**
-- **Operators**
-- Again, we do not repeat the operator descriptions.
function IS_A_COMMON_WORD(A_WORD : WORD)
                    return BOOLEAN;

procedure ENTER
  (A_TITLE   : TITLE;
  A_KWIC_INDEX : KWIC_INDEX;
  A_POSITION   : POSITION);
function WHICH_POSITION
  (WORDS   : THAT_WORD_AND_
                : SUBSEQUENT_WORDS;
  A_KWIC_INDEX : KWIC_INDEX) **return** POSITION;
end KWIC_STUFF;

with TITLES_AND_WORDS; **use** TITLES_AND_WORDS;
with KWIC_STUFF; **use** KWIC_STUFF;

procedure PRODUCE_KWIC_INDEX
  (THE_TITLE_LIST   : TITLE_LIST;
  THE_KWIC_INDEX : **out** KWIC_INDEX) **is**

THE_TITLE   : TITLE;
THE_WORD   : WORD;
THE_POSITION : POSITION   -- This object is not used
                       -- and can be deleted.
MORE_TITLES : BOOLEAN := TRUE;   -- This did not
MORE_WORDS : BOOLEAN := TRUE;   -- appear in the
                       -- original list of
                       -- objects.
begin
  CONSIDER_FIRST_TITLE
      (THE_TITLE, THE_TITLE_LIST, MORE_TITLES);
  while MORE_TITLES **loop**
    CONSIDER_FIRST_WORD(THE_WORD,
                  THE_TITLE, MORE_WORDS);
    while MORE_WORDS **loop**
      if not IS_A_COMMON_WORD(THE_WORD) **then**
        ENTER(TITLE,
            KWIC_INDEX,
            POSITION(REMAINING_WORDS
                   (THE_TITLE), KWIC_INDEX));

```
    end if;
    CONSIDER_NEXT_WORD(THE_WORD,
                    THE_TITLE, MORE_WORDS);
    end loop;
    CONSIDER_NEXT_TITLE(THE_TITLE,
                    THE_TITLE_LIST, MORE_TITLES);
    end loop;
end PRODUCE_KWIC_INDEX;
```

The embedding of ENTER in the inner loop of the program apparently precludes the use of an alphabetization technique that requires the elements to be alphabetized to be available all at once. This is not a serious problem. Should one determine that an all-at-once alphabetization technique is preferable it can easily be accommodated. One would define an ALPHABETIZE operator on KWIC_INDICES that performed the desired all-at-once alphabetization. One would then redefine ENTER to enter the titles in nonalphabetical order. The new ALPHABETIZE operator would be applied to the entire index after the outer loop. The result is that the original algorithm (that has the alphabetization take place incrementally) has been changed. But the data types identified by the common nouns and used by the program remain the same.

11. DISCUSSION
11.1. How Automatic is the Transformation Process?
From the process we followed, one might imagine that it can be a purely mechanical procedure to transform an informal strategy into a formal program. That is not at all the case. A great deal of understanding and background knowledge were required for the transformations we made. A computer program that can take an informal strategy expressed in English and transform it automatically to an executable program is still a long way off.

11.2. Must Program Design be a Top–Down Process?
The procedure we followed seemed to be purely top–down. Most problems are not quite so simple. The basic reason these problems lent themselves to such purely top–down solutions is that the solutions were built on well-known, rigorously defined, preexisting models as bases. The solution to the calendar problem was built on the base of the standard Gregorian calendar. The solution to the KWIC problem was built on our intuitive understanding of titles, words, etc. There is no question that we can implement the operators as specified in the packages for these models.

Most computer programs are not built on such convenient, preexisting base models. In most cases, the top level has to be defined in terms of lower level concepts that themselves have not yet been fully defined. In such situations, it is not always clear that the lower level concepts can be defined in a way that is most convenient for the top level. Sometimes the lower level model requires for its internal consistency that the higher level conform to certain constraints.

Even in the simple calendar problem we found we had to add a data type YEAR (for use in the operator NUMBER_OF_DAYS_IN_MONTH) that did not originally appear in our informal strategy. There is nothing in the original solution strategy to suggest the need for the data type YEAR in the final problem solution. Nonetheless, the data type YEAR did turn out to be necessary.

This is only a minor example of how a lower level model can force changes in a higher level program. This occurred in a situation where the lower level model was well-defined and known before we began. The problem, of course, is that the notion of month does not have built into it all the details that

we eventually require of it. To completely define the operator NUMBER_OF_DAYS_IN_MONTH, the notion of year is also needed. This is typical of "real world" examples in which apparently well-understood notions turn out to be not so well-defined.

This also illustrates one of the differences between common nouns and data types and why common nouns are an extension of the usual understanding of data types. MONTH is a common noun; one has a fairly clear understanding of what a MONTH is, even though one is not necessarily conscious of the need for the data type YEAR in defining all its characteristics. A data type is defined exactly as a composite of its values and operators, so it is impossible to have a data type without its operators completely defined.

In the more common situation, when one is developing a program top–down in terms of a less familiar lower level model that itself is not yet defined, it is very unlikely that the top level can be defined once and for all and never change. It is much more likely that the lower level model will force changes in higher levels.

The likelihood of having to return to higher levels while developing lower levels should not discourage modular thinking. It is very important that the top level and, in fact, every level, be defined in a consistent and self-contained manner. Every module should be defined in its own terms and in terms of data types and operators that are meaningful to that module. The concepts captured by the common nouns appearing in an intuitive description of each level will usually suggest a good initial collection of abstractions around which appropriate modules may be built.

11.3. Final Remark
The main lesson to be learned from this exercise is that programs can be developed in terms that match intuitive data types and operators. The concepts used to understand a problem originally are generally the best concepts to use to write the program to solve the problem. This is not to say that the first idea one has is necessarily the best approach to take. It is often the case that one's original idea for an algorithm can be greatly improved. Nonetheless, it is usually a good idea to identify and formalize the intuitive concepts, that is, data types, with which the program is concerned.

REFERENCES
1. Abbott, R. J. An Integrated Approach to Software Development. Addison-Wesley, (in preparation).
2. Booch, G. Software Engineering with Ada. Benjamin Cummings, 1983.
3. Dijkstra, E. The humble programmer. Comm. ACM. 15, 10 (Oct. 1972).
4. U.S. Department of Defense. Reference Manual for the Ada Programming Language. MIL-STD 1815, 1980.
5. Guttag, J. V., Horowitz, E., and Musser, D. R. Abstract data types and software validation. Comm. ACM. 21, 12 (Dec. 1978) 1048–1062.
6. Hoare, C. A. R. An axiomatic approach to computer programming. Comm. ACM 12, 10 (Oct. 1969) 576–580.
7. Liskov, B. H. and Zilles, S. Specification Techniques for Data Abstractions. IEEE Trans. Software Engineering. SE-1, 1, (March 1975) 7–19.
8. Parnas, D. L. On the criteria to be used in decomposing systems into modules. Comm. ACM. 15, 12 (Dec. 1972) 1053–1058.
9. Quine, W. V. Word and Object. MIT Press, Cambridge, MA 1960.
10. Wirth, N. Systematic Programming: An Introduction. Prentice–Hall International, Englewood Cliffs, NJ, 1973.