Principles of Package Design

Bertrand Meyer
Electricité de France (EDF)

SUMMARY: Subprogram packages are groups of related subroutines used to extend the available facilities in a programming system. The results of developing several such packages for various applications are presented, with a distinction made between external and internal design criteria—what properties packages should offer to their users and the guidelines designers should follow in order to provide them. An important issue, the design of reusable software, is thus addressed, and the concept of abstract data types proposed as a desirable solution.

1. Introduction

For several years some of us at EDF have been writing software tools of general applicability. The term *Atelier logiciel* (software workshop) has been used to describe our team's activity. The tools which have been constructed and distributed differ widely in their nature and mode of utilization. An important category is that of subprogram packages. A subprogram package is a group of routines which may be called by any program; its purpose is to provide a means of performing tasks in some domain of application which the available programming language does not directly address.

Examples of subroutine packages which we have developed during the past three years include those listed in Figure 1. Working on these packages, we have gained various insights. Our aim here is to convey some of these to other practitioners who may be confronted with similar problems. No breakthrough is claimed; our techniques are mostly standard. We feel, however, that their presentation and a discussion of the software engineering methods used in the design of our packages may be helpful to practicing programmers working in an "industrial" environment.

In Section 2, we describe our environment, a large scientific computing center, and underscore the need for subprogram packages in relation to other kinds of software tools. Section 3 is a detailed discussion of external design criteria, i.e., how packages should appear to the outside world. Section 4 presents our methods for internal design, i.e., implementation to fulfill the requirements of the preceding section; the gist of our approach is that it considers a package the implementation of one or more *abstract data types*. Section 5 concludes with some reflections on the scope of our experience.

Since naming conventions form an important part of our discussion, we have, throughout the text, translated the French words and abbreviations appearing in subprogram names. The package names themselves have been preserved.

2. Why Subprogram Packages

The ideas presented here certainly reflect to some extent the fact that our computing center is geared toward scientific, mostly Fortran programming; and, to a lesser one, that it uses three IBM computers (370–168, 3033, 3081) under MVS, to which a Cray–1 has recently been added.

The first question the reader may ask is why we concentrate on collections of subprograms. Our aim is to extend the range of facilities offered by the existing language. There are at least four other solutions:

(1) convincing users to switch to a better or more powerful language;
(2) writing JCL procedures;
(3) writing conversational procedures;
(4) designing special-purpose preprocessors.

CR Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General–*standards*; D.2.2 [**Software Engineering**]: Tools and Techniques–*modules and interfaces, software libraries, user interfaces*; D.2.7 [**Software Engineering**]: Distribution and Maintenance–*documentation, extensibility*; D.3.3 [**Programming Languages**]: Language Constructs–*abstract data types, modules, packages.*
General Terms: Design, Documentation, Languages, Reliability.
Additional Key Words and Phrases: Reusable software, software tool, Fortran.
Author's present address: B. Meyer, Electricité de France (EDF)—Direction des Etudes et Recherches, 1, avenue du Général de Gaulle, 92141 Clamart, France.

Briefly, we shall discuss why these choices are not always satisfactory.

Solution (1) is certainly the ideal one. However, the sad fact is that most programmers in industry use first-generation languages and are unlikely to try another one. If your aim is to produce tools that will be used, you had best conform to the majority rule. (An even sadder fact, as we shall see in Section 4, is that the tool writer is usually barred from using modern languages because of technical constraints.)

Solutions (2) and (3) (batch or conversational procedures) are adequate for tools intended for "end users", but not for tasks whose execution is initiated by programs.

Solution (4) (preprocessors) may seem attractive but there are many drawbacks involved. One is that it may lead to the proliferation of preprocessors serving various purposes, which will not be, as a rule, mutually compatible. As an example, consider the case of a Fortran programmer who wishes to use the control structures of "structured programming. His programs output results to various graphic devices, and they require that some arrays have dynamic bounds (i.e., the bounds are read on a file before processing begins). Many preprocessors, such as Ratfor [5], are available for the first purpose; others, such as Fortran 3D [11], serve the second one (note, however, that the current release of the latter product uses the subprogram package formula); still others exist for the third requirement. The input languages for these preprocessors will, in general, use wildly different conventions. Their treatment of errors will not be the same. Some of them, in generating Fortran code, will delete comments, while others will recognize comments under a certain predefined syntax as directives. Their combined use will thus be very difficult and, in many cases, impossible.

Preprocessors present another well-known problem. Often simple-minded, they do not provide all the services expected from a well-engineered compiler (cross-references, symbol tables, data flow analysis, useful error messages, source level optimization). They usually have no associated run-time systems, let alone debugging aids. Since they generate code in existing programming languages, they rely on the associated facilities. This makes run-time errors a source of distress: they must be traced back through a program-generated program, which is hardly more readable than the object code produced by a compiler.

Ensorcelé—	free-form input and output
Chronos—	time measurement
Textes—	text manipulation
Axédir—	direct-access file management
Gescran—	full-screen programming
Tri—	internal sorting

Fig. 1. Packages and Their Aims.

Initialization and Termination	
CALL ASKGGE (answer)	May I use full screen? (yes, if answer = 0)
CALL LEAGGE	Leave full-screen mode.
Defining Screens and Creating Windows	
CALL DEFSGE (*ns*)	Define *ns* as the name of a screen.
CALL MXLSGE (*n*)	Set to *n* the maximum number of window lines per screen.
CALL CREWGE (*nw, ns, il, ir, iu, id*)	Create window *nw* in screen *ns* with *il, ir, iu, id* as coordinates.
CALL DELWGE (*nw*)	Delete window *nw*.
CALL BRIWGE (*nw, b*)	Assign brightness *b* to window *nw*.
CALL PROWGE (*nw*)	Make window *nw* protected.
CALL FREWGE (*nw*)	Make window *nw* free (unprotected).
CALL CAPWGE (*nw*)	From now on, convert letters in window *nw* to capitals.
CALL ASIWGE (*nw*)	From now on, leave any character in window *nw* as it stands.
Changing or Examining the Internal Image	
CALL REPWGE (*nw, tabcha*)	Replace contents of window *nw* with *tabcha*.
CALL BLAWGE (*nw*)	Fill window *nw* with blanks.
CALL ASSSGE (*nst, nss*)	Assign value of screen *nss* to screen *nst*.
CALL BLASGE (*ns*)	Fill all unprotected windows of screen *ns* with blanks.
CALL NBCWGE (*nw, n*)	Assign to *nm* the number of changes to window *nw* since the last input operation.
CALL EXAWGE (*nw, tabcha*)	Assign to *tabcha* the current contents of window *nw*.
Input and Output (Affecting the External Image)	
CALL WRISGE (*ns*)	Display screen *ns* on the terminal.
CALL REASGE (*ns*)	Input screen *ns* from the terminal.
Manipulating the Cursor and Function Keys	
CALL POSCGE (*nw, nline, ncol*)	Position the cursor in window *nw*, at position [*nline, ncol*].
CALL EXACGE (*nw, nline, ncol*)	To what position [*nline, ncol*] was the cursor in window *nw*? ([*0, 0*] if not in window)
CALL EXAKGE (*ns, n*)	Assign to *n* the number of the function key used to send the screen contents.
Typed Input-Output (Interface with Package Ensorcelé)	
CALL UNIOGE (*nw*)	Direct subsequent output to window *nw*.
CALL UNIIGE (*nw*)	Obtain subsequent input from window *nw*.

Fig. 2. Reference Sheet for Gescran.

Additionally, it should be noted that preprocessors only add surface improvements to Fortran. They usually do not provide remedies for this language's intrinsic limitations with regard to data structuring, dynamic allocation, pointer variables, intra and inter-routine type checking, recursion, etc.

Subprogram packages do not suffer from these defects, although, admittedly, they raise other problems which we discuss in the next two sections. To the potential user, they offer a very neat way of enriching the existing programming language with new instructions, implemented as subprogram calls.

3. External Design Criteria

A subprogram package is a collection of mutually related subprograms. Just how they should be "related" to each other will be studied in more detail in Section 4. For the moment, we turn to an important question: How should these subprograms be presented to their potential users? This problem is vital, especially in light of the fact that programmers are often reluctant to invest the effort necessary to learn a new methodology. They will not be lured into using our packages unless some very attractive arguments convince them to do so.

In the following subsections we shall list those desirable qualities which our packages should possess and explain exactly how these design criteria—namely, simplicity; self-restraint; ease of use; homogeneity, safety—were met.

3.1 Overall Simplicity

In the area of simplicity our central thesis was that most programmers would not use a subprogram package if it required constant reliance on a reference manual. Although we did insist that users read part of the manual at least once, our ideal was that they should then be able to employ a package for standard applications without further reference to any written document. In practice, we have not succeeded in reaching this goal completely, but we have nevertheless succeeded in concentrating all the necessary information for normal use of a package on a single page. This we consider a mandatory requirement. For an example, see the reference sheet for the package Gescran as outlined in Figure 2.

The most important aspect of our approach is that we do not try to write complex packages providing a wide range of services and satisfying all users' fantasies. Instead, we concentrate on a careful study of user needs and strive to offer simple and efficient answers to the most important of them. Of course, deciding which issues are the most important is a design decision since often user needs are either unexpressed or, if expressed, require much work to be transformed into realistic specifications.

3.2 Self-Restraint

Our subprograms are called by other programs or subprograms: they are not directly concerned with solving "interesting" problems, but rather with performing general utility tasks. The application program/subprogram package/system hierarchy is pictured in Figure 3; other levels may, of course, exist. We shall refer to the programs which call our subprograms as *application programs*; on the other hand, *users* will be those individuals (or programs) who (which) run application programs. (These terms, especially the latter, are two of the most misused in data processing; we shall strive to use them precisely.)

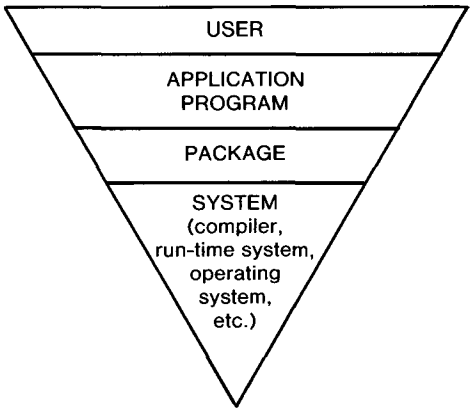

Fig. 3. Hierarchy of Programs and Program Users.

Self-restraint is necessary because there is at least one level, that of an application program, between users and our subprograms. The latter must thus be as invisible to users as possible. This is especially important in connection with errors (Section 3.5).

3.3 Ease of Use

Documentation

Documentation is organized in terms of simplicity, ease of use, and homogeneity. All packages are documented by manuals with the same structure, as shown in Figure 4.

Order of Arguments

One key to ease of use is consistency of design. This criterion becomes even more crucial as new packages are employed and the number of available subprograms grows. It requires that a set of regular, coherent conventions be strictly observed for all distributed products.

```
Foreword ("How to Use This Manual")
    Section 1—Introduction
            2—Individual Subprogram Description
            3—Restrictions and Caveat
            4—Examples
            5—Notions on the Implementation
Appendix A—Error Messages
            B—List of External Names
            C—Portability
            D—Performance
            E—Control and Data Flow Graph
            F—Quick Reference List (last page)
```

Fig. 4. Structure of the Manuals.

An important area requiring a homogeneous policy is parameter order. In a language environment not providing for key word parameter transmission, actual arguments to any subprogram must be given in a fixed order, which matches that of formal parameters for the subprogram. Package users must thus know this order; such a constraint often becomes a source of annoyance and errors. It is therefore desirable that the package designer adhere to some convention.

For example, in the Textes package, which allows character string manipulation using pseudo-string variables that appear to the compiler as integer variables (see Section 4.6), the syntax of some typical calls would be what is seen in the box below. *itext*, *jtext*, and *ktext* are pseudo-variables, *i* and *j* integers.

The order of arguments decided on here was the following: in assignments the destination should always precede the source. This is consistent with the syntax of most programming languages:

$$A := f(B, C, \ldots)$$

Moreover, since the package's aim is to provide the equivalent of a "string" data type as it exists in, say, PL/I, the chosen order aims to imitate the syntax of languages which do offer operations on this type; e.g., CALL CNCTTX (*itext*, *jtext*, *ktext*) follows the PL/I pattern *itext* = *jtext* || *ktext*.

The rule of consistency in the order of arguments may conflict with other, equally important criteria concerning the homogeneity of design. For example, in the Axédir package for direct-access file management, there is a read routine whose call has the following form

CALL REAFDA (*file-id, target, record-number, error-indicator*)

This conforms to the "destination first" rule, although the file identifier comes before the target for reasons of consistency with the rest of the package. For the write routine, however, we chose the syntax

CALL WRIFDA (*file-id, source, record-number, error-indicator*)

since we thought it would be easy to remember that corresponding arguments occupy exactly the same position in both operations, "target" for read and "source" for write being symmetric. The destination first rule is thus violated by WRIFDA.

3.4 Homogeneity

Number of Arguments (operands and parameters)

The question of arguments involves simplicity as well as homogeneity. Not only should arguments appear in a carefully chosen order, but the number of them should also be small if programmers are to remember the calling sequence.

Of all the subprograms listed in Figure 1, 71 percent have zero, one, or two arguments, and less than 4 percent have more than four (a function result being counted as an argument). The maximum number of arguments is six.

Requiring short argument lists has an immediate consequence: since any means of data transmission between an application program and a package subprogram other than argument passing (such as explicit COMMON block sharing) is banned, every subprogram may perform only a well-defined single task. In our case, this property became another motivation for requiring short argument lists, rather than a consequence of this requirement. It is indeed integral to our design philosophy (see Section 4).

Such an approach has interesting practical consequences which distinguish our packages from many commercially available ones. Let a subprogram, say *f*, be used to implement an operation with a certain number, say *n*, of *operands*. It is often the case that several operating modes are available, described by a certain number, say *m*, of *parameters* or *options*. Quite commonly, *n* is small, but *m* may be large and will grow as users request new refinements.

At this point, the reader may ask for a precise definition of the distinction between parameters and operands. Although the difference is in many cases intuitively obvious, an absolute definition does not exist. Rather, the distinction should be thought of as design decision which the designer bases on the following guidelines:

—The number *n* of operands should remain small.

—The system should be able to set default values for parameters.

—During the package's evolution, as parameters are added (or removed), the specification of operands for any single subprogram must not be changed.

Thus, the distinction between parameters and operands is partly a pledge made by the designer with respect to the future of the package.

There are three ways of specifying a subprogram *f* with both operands and parameters:

CALL CRETTX (*itext*)	(CREate a Text variable) Create a new string variable, of name *itext* (pseudo-declaration).
CALL CNSTTX (*itext*, '*xyz . . .*')	(CoNStant Text) Assign the character string '*xyz . . .*' to the string variable *itext*.
CALL CNCTTX (*itext*, *jtext*, *ktext*)	(CoNCatenate Text) Assign to *itext* the value of *jtext* concatenated to that of *ktext*.
CALL SUBTTX (*itext*, *jtext*, *i*, *l*)	(SUBText) Assign to *itext* the value of the substring of *jtext* starting at position *i*, with *l* characters.

(a) Include all necessary operands and parameters in every subprogram call, as in

CALL $f(opnd_1, \ldots, opnd_n, parm_1, \ldots, parm_m)$.

(b) Include only operands, as in

CALL $f(opnd_1, \ldots, opnd_n)$

and provide other subprograms, one per parameter, to set the values of parameters, in the form

CALL $setval_i(parm_i)$

with the understanding that the ith parameter will remain set to the value $parm_i$ until a new call to $setval_i$.

(c) Use a mixed-mode approach, with some parameters included in the calls to f and others separately.

Throughout our packages, we adhered to the second approach (b), which we find preferable for two basic reasons:

(1) It allows the package designer to set default values for all parameters, thus freeing the user from providing arguments corresponding to options not of primary concern.

(2) Including parameters in the operation invocation inevitably leads to problems as the package evolves: although operands usually do not change if the initial design is sound, requests for new parameters will appear. We have experienced this phenomenon over and over again. For example, users of Ensorcelé (free-form input and output) requested new facilities for output formatting. To meet their request, we added a "color" parameter to the Gescran subprograms when color displays became available. Had we included parameters in the calls, all the calling programs would have had to be changed, making it very difficult to entice anyone into using our programs afterwards. Thanks to our seemingly drastic policy, we have so far been able to avoid such a situation.

Note that the use of a language allowing subprograms to have both positional and key word arguments (such as Ada) would solve the problems inherent in situation (1), but not (2).

One may object that our technique increases the size and external complexity of packages since there will be one subprogram per parameter per operation. This does not worry us too much because there is not much difference in added complexity between a new subprogram, on the one hand, and a new argument to an existing subprogram, on the other.

Another possible drawback is that application programs will contain many subprogram calls when they require nondefault options. For example, if a user wishes to output a real number X in a particular format, using Ensorcelé, the sequence of instructions could be as long as the one listed in the box on this page.

Although such code may seem horrendous to experienced programmers, we find it quite acceptable (and have even come to like it!). It is really very readable since every call has a clearly stated single purpose. Also, remember that parameters remain set until explicitly changed so after initialization, there will usually be fewer calls to the parameter-setting routines (unless, of course, the user program wishes to often change options).

All in all, we feel our strictly functional approach, with a clear distinction between operands and parameters (and between operation and parameter-setting subprograms), is very helpful in the design of coherent, easy-to-use, and simply maintained packages.

CALL SAVPAR	Save the current values of Ensorcelé parameters.
CALL EXPON (5)	Real numbers will be output using the exponent (E) format if their absolute values are not in $]10^{-5}, 10^{+5}[$.
CALL BLANKS (3)	Output items will be separated by at least three blanks.
CALL ZONE (9)	Items will be justified to the right in zones of length 9 (or a multiple thereof if they do not fit).
CALL NBRDIG (8)	At least eight significant digits should be printed.
CALL PUTZER	Trailing zeros should be written (default: blanks).
CALL WRIREA (X)	Write X.
CALL RESPAR	Restore previous parameter values.

External Subprogram Names

An important component of homogeneity as well as the aforementioned criteria of ease of use is how external subprogram names are chosen. This issue is a delicate one (which we had not well understood when we started our work) because of four conflicting requirements:

(1) the desire to provide mnemonic names, as expressive as possible;
(2) the need to avoid possible conflicts with names of subprograms or data segments in the application programs;
(3) the need for a coherent set of naming conventions, which grows with the number of available packages and subprograms (and the size of the programming team);
(4) for subprograms callable from IBM Fortran, the tight 6-character limit.

At the outset, we had, with clarity our goal, concentrated on the first requirement. The reader may have noted names such as BLANKS and ZONE in the Ensorcelé example cited in Section 3.3. Inevitably, this led to conflicts with names chosen by application programmers and we had to adopt a more balanced strategy. All of our current subprograms have 6-character names with the following structure:

—Three letters which are an abbreviation for a "verb" denoting the action to be performed, e.g., REA for read, SET for set;
—One letter indicating the type of object to which the action applies, such as I for integer, C for cursor;
—Two letters which are a code assigned to the package, e.g., GE for Gescran.

Thus, the subprogram positioning the cursor somewhere in Gescran has the name *SETCGE*.

Using this technique, we have been able (with some care) to avoid name clashes. Additionally, the method is simple to explain in the package manuals so the name may be considered mnemonic for the application programmers.

3.5 Safety

Treatment of Errors

An important but difficult issue is that of errors: How should a general-purpose routine react in an error situation?

First, we shall define precisely what an error is in the context of our packages. A subprogram in such a package is intended to complete some *actions* and/or to compute some *values*. An error arises when the subprogram detects that an action cannot be performed or that a requested value does not exist. In either case, it means the subprogram is able to determine the fact that a certain element does not belong to the domain of a certain function (which is part of the subprogram's abstract specification).

The possibility of an error made in writing the subprogram being ruled out, the cause of the error may be either of the following:

—The user has provided illegal arguments to a subprogram.
—Some well-founded request cannot be satisfied because of external conditions (e.g., dynamic memory allocation fails since no more space exists).

What policy should the package writer adopt in regard to such errors? There are two conflicting requirements: safety and self-restraint.

(1) Safety implies that no operation not conforming to the application programmer's intent and, in particular, no modification of the application program's state other than those explicitly provided for in the package's manual should ever be performed. Additionally, the application program must be able to find out about the error and take any corrective action it wishes.

(2) The need for *self-restraint*, on the other hand, stems from the fact that it is very difficult to decide what action to take on the sole basis of what is known to the subprogram (the same situation is experienced by, say, the writer of a lexical analyzer in a compiler). It suggests that the package should be able to make a reasonable correction, without unnecessarily bothering the calling program, let alone causing a system interrupt.

One way to ameliorate the problem of errors is to avoid illegal arguments by enforcing as few restrictions on subprogram calls as possible (which in effect means expanding the specifications to include most "error" cases as peculiar but legal ones). Because of such a policy, we experienced very few error cases in our first packages and were able to adopt a rather haphazard approach to error treatment (see the "error-indicators" in the calls to the Axedir subprograms in Section 3.3).

Recently, we have arrived at the following approach. A small package, called Errare, which is comprised of only three subprograms has been designed:

(1) CALL RECEER (*n*, '*message*'), RECE standing for RECord Error, sets a global error indicator to *n* and outputs the message along with other information, in particular the operating chain (in order to avoid avalanche effects, a shorter text is output whenever *n* is equal to the previous error indicator).

(2) INDEER (0), an integer function with no arguments (a dummy argument is required in Fortran 66), returns the value of the global error indicator (as set by the last call to RECEER; zero if none).

(3) CALL SETUER (*n*), SET Unit, directs subsequent message output performed by RECEER to output unit number *n* (recall that in Fortran, I/0 devices are designated by integers between one and 99). If SETUER is not called, error output will be printed on the standard output file.

With these subprograms, a package subprogram takes the following course of action when it detects an error.

—Record the error number and output a message with RECEER.
—If an action was requested, do not do anything.
—If a value should have been computed, then two subcases arise: when a sensible approximation exists, use it as a substitute; otherwise, return a value chosen to be as "out of bounds" as possible (e.g., a negative integer if an address was requested).

This technique seems both self-restrained and safe. It is self-restrained because INDEER is a public function. Thus, if the application programmer wishes to correct errors possibly occurring in a package subprogram, he can do so by testing INDEER after the call; the programmer will thereby remain in full control of all events since the package itself does nothing abnormal except outputting a message. The technique is safe because it guarantees that no illegal action will be performed by the subprogram. On the other hand, if no reasonable value can be computed, the result will be so absurd that it will inevitably lead to program abort shortly after the call unless the application program regains control with INDEER. It is certainly much better to provoke a "negative address" error than to allow the program to work on an erroneous but physically meaningful address.

The use of "abnormal" values, such as negative numbers when an address or array index would have been required, is only possible because of the lack of strong type checking in Fortran. The transposition of this technique to languages with stronger type requirements requires the presence of an *undefined* value in every type. This condition is met by languages like Algol W and Simula 67 in which all programmer-

defined types are pointer ones with a special empty value (called *null* or *none*) as one of their elements. No such possibility exists in Pascal or Ada whose record types, for example, do not possess a void value.

One advantage of our method is that the treatment of errors does not interfere with other criteria. In particular, in terms of argument lists, the external specification of package subprograms does not have to be changed. Better general solutions are hard to find, short of an exception facility like those in PL/I or Ada.

3.6 Functions vs. Subroutines

Almost all of our subprograms are subroutines (actions) rather than functions. Using a function may seem preferable in the case of a subprogram returning a single value and having no side-effect; the reader may have wondered while reading about the Textes package (Section 3.3) why we used a subroutine to compute the concatenation of two strings. Indeed, if we want to output the concatenation of *jtext* and *ktext*, we must write what appears in the box above instead of the much more natural

CALL PRNTTX (*fcnttx* (*jtext*, *ktext*))

where *fcnttx* would be a function returning the concatenated string.

We found three objections to using functions.

(1) In many systems, including ours, Fortran functions cannot be called from Cobol programs (whereas subroutines can). Since we do have a few Cobol users, subroutine interfaces must be written anyway.

(2) A function type must be declared in the calling program, except when it is integer or single-precision real and follows the Fortran default rule (which eliminates logical, double precision, and the Fortran 77 character type). This is a source of errors in systems with no checking at link or load time.

(3) An important issue in deciding whether to express the same semantics as $x = f(a, b, \ldots)$ or CALL $f(x, a, b, \ldots)$ is that only the latter

```
INTEGER itext
.....
CALL CRETTX (itext)          pseudo-declaration of string variable
.....
CALL CNCTTX (itext, jtext, ktext)   assign to itext the concatenated string
                                    output
CALL PRNTTX (itext)
```

construct gives the subprogram writer access to *all* the operands involved, including x, which may be needed in order to make f safer and/or more efficient. Both safety and efficiency were at stake in the choice made for the Textes package. On the one hand, since string operands are integers for the compiler, our subprograms must be able to check whether both sources and target have been correctly pseudo-declared, thus avoiding dangling run-time references. On the other hand, the package uses quite an elaborate memory management algorithm [7] and will save a lot of space when *itext* is the same string variable as *jtext* or when the previous allocation for *itext* is greater than or equal to length (*jtext*) + length (*ktext*).

In view of these factors, we only use Fortran functions for integer functions giving the value of some attribute of an object. This occurs in the sense of Section 4.2 (that is, an "accessor function" as defined in connection with abstract data types). For example, the length of string *itext* is denoted by *LNGTTX* (*itext*).

4. Internal Design Techniques

4.1 Framework

In the previous section we described our basic aim: to provide our products' potential users (the application programmers) with packages whose external appearance is sound and coherent. The key to success is, of course, that these properties be matched by the stability and consistency of internal design. As Jackson [3] remarked about early attempts to define modular programming, words like "functional integrity" are not very useful as practical design guidelines as long as they remain unsupported by more technical definitions of the methods used. The concept which we have found most fruitful as a design base for sound subroutine packages is abstract data types, a notion now well-established in academic and research circles although practically unheard of by most practicing programmers.

4.2 Abstract Data Types

An abstract data type is the formal definition of a data structure or class of data structures, as characterized by purely functional properties. The definition of an abstract data type T comprises three parts:

- a list of *domain names*, one of which is T;
- a list of *function names* with associated functionalities, i.e., domains of the arguments and results (at least one of these domains must be T for every function); these functions are the abstract representation of the operations available on the type;
- a list of logical *assertions* on these functions, which describe the operations' formal properties.

A definition comprised of these elements is a formal *specification* of the data type.

An *implementation* of an abstract data type is a set of data definitions and subprograms operating on the data defined, such that each datum's type (with the ordinary meaning of the word "type" in programming languages) is associated with one of the domains in the abstract data type's definition. Each subprogram corresponds to one of the functions and satisfies its functionality requirement with respect to input and output arguments. The values of these arguments satisfy the assertions for every call of the subprogram.

Some have argued that a good way, perhaps the best, to construct truly modular programming systems is to organize them as sets of abstract

data type implementations. This claim is supported by practical evidence [13].

It and other reasons explain why we have used abstract data types as the model for our packages. In fact, every one of our packages is the conscious implementation of one or more abstract data types. In particular:

— The Textes package implements the "text" or "string" type with operations like the creation of a constant text, the extraction or modification of the *i*th character, or concatenation.

— The Chronos package implements the "time counter" concept.

— The Axédir package implements the external array type with "initialize," "read," and "write" as operations.

— The Gescran package implements the "page" (or "screen") and "window" abstract data types with operations like "define window in screen," "write into window," or "visualize screen."

It is therefore not surprising that the main design choices we encountered in implementing packages are conveniently expressed in terms of abstract data types. In the following subsections, we study some of the most important, namely: linguistic issues; heirarchical design; static *vs.* dynamic instanciation; information hiding.

4.3 Linguistic Issues

The programming language for writing a package should offer a structure corresponding to the schema just presented. This is indeed the case in many recent languages. Foremost among these, from the practitioner's point of view, are the pioneer, Simula 67 [1, 8], and the youngest, Ada [2]—the former because of its availability on a variety of machines, the latter on account of its intended wide circulation.

These languages, like their relatives (Lis, Clu, Alphard, Euclid, Mesa, Modula), include a program structure ("class" in Simula and "package" in Ada) with three categories of elements: data definitions, subprogram declarations, and statements. Such a structure may be used to implement an abstract data type (or an object of such a type, see Section 4.5); its three components correspond to data representation, operations, and initialization, respectively. Given an instance, A, of a class/package and x as one of its components (subprogram or data element), an external module which is entitled to "use" A may reference x. This is done either with a "dot notation", $A.x$, or directly by its name, x, provided that the external module has "acquired" A in some fashion (**inspect** A in Simula, **use** A in Ada) and there is no name conflict.

This kind of solution is very convenient, both from the package writer and application programmer's point of view. The former designs and implements the package as a single module, separately compilable and verifiable: all the relevant information is concentrated in a single, coherent entity. The application programmer, when requesting a function performed by the module, simply supplies the names of the module and the function.

Unfortunately, it is usually impossible to write subprogram packages in such a language, even if one is available. Although "first-generation" languages like Fortran and Cobol and the assembly languages for most machines are geared toward a very simple, static allocation policy, newer languages (including not only "modular languages" but also PL/I, Algol W, and Pascal) require a much more ambitious memory management scheme, usually with a stack and a heap, the latter being subject to garbage collection. Therefore, even with well-engineered language systems permitting separate compilation and linking with modules written in other languages, the system for the more elaborate language must exercise control at run-time. For most systems, this precludes the use of such a language for writing subprogram packages since the latter must be accessible to any program.

The tool writer is thus placed in a very frustrating situation. He knows the right language(s) in which to write a subprogram, but he remains unable to use it. We, for instance, have a very good Simula system [9] but must resort to Fortran for subprogram packages, with all its drawbacks: no data structure other than the array, no control structure other than the **If** and **Goto**, no pointer variables, no dynamically created elements, no parameterized-dimension arrays, no recursion, and, of course, no "class" or "package" structure.

4.4 Hierarchical Design

In order for each element of a package to remain simple and understandable, it is necessary that the package's structure consist of several layers in all but the most trivial cases. For packages seen as implementations of abstract data types, this means such an implementation will use objects belonging to other types, also defined abstractly, i.e., used through their properties rather than representation. Thus, a package is generally implemented as a hierarchy of types. Such a hierarchy is illustrated in Figure 5. Ensorcelé 1 (output) appears as a means for manipulating a *stream* of "printable" objects, which is represented using the concept of unbounded *character string*, itself implemented as a sequence of *lines*.

Out of the many advantages of this approach, two are worth noting. First, it allows the designer to push down all machine- and system-dependent elements to the lowest levels of the hierarchy, thus increasing portability (for example, Gescran was built for the IBM 3270 terminals, but only a few subprograms must be recoded for other similar devices). Second, it lends itself to top-down design, which, as Wirth pointed out [12], should apply to data as well as control.

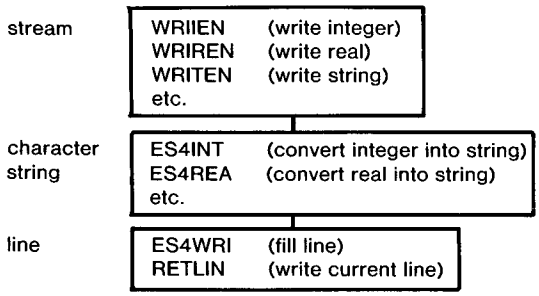

Fig. 5. Hierarchy of Types for Writing (Ensorcelé 1).

4.5 Static vs. Dynamic Allocation

A package implementing an abstract data type may provide one of the following:

(1) one object of the type;
(2) a fixed number of objects of the type;
(3) an unlimited number of objects, within the limits of the available space at execution time.

Solution (1) provides for the implementation of what may be called an "abstract object" rather than a type. It is used, for example, in Ensorcelé which acts on a single stream of objects.

Solution (2) is quite natural in Fortran because of the arrays' static dimensions. For example, one package similar to our Textes in terms of the services offered [10] provides a fixed number of text variables, corresponding to the size of an array in a COMMON block. Of course, this often results in unpleasant repercussions since the limit may appear too large (entailing undue space use) or too small (requiring recompilation of separate versions of the package). We have seldom used this technique; an example is Chronos, which sets an absolute limit of 100 time counters.

Solution (3) comes closest to what is offered in languages providing user-defined nonstatic types. Every object of the type needed in the application program must be explicitly created by it (*new* statement in Simula or Pascal). This is the most powerful solution; its main drawbacks within our framework is that a few non-Fortran (or nonstandard) routines for dynamic memory allocation must be used. Packages like Gescran, Axédir, and Textes provide an unlimited number of instances (character strings in the first, files in the second, "screens" and "windows" in the third).

4.6 Information Hiding

One of the main goals of the abstract data type approach is a clear separation between what is visible to application programmers and what remains private to the package designer. The latter category should include all elements dependent on nonessential hardware, system, implementation, or design peculiarities. We have found two techniques useful in enhancing this property: the careful choice of names and the use of pseudo-variables.

Internal names are chosen so as to seem mnemonic only to team members. Like external names (see section 3.4), they follow a regular pattern and make collisions unlikely.

The notion of a pseudo-variable is more important. In the case of packages offering an unbounded number of type instances, the individual objects must be nameable by the application programs, although Fortran does not offer a declaration other than for standard types. The solution is to declare the objects using names which appear to the compiler as those of integer variables. Actual "declaration" will then be effected by a call to an instantiating subprogram. Such pseudo-variables were used in the Textes example cited in Section 3.3.

Internally, the integer variable will usually contain a pointer to the location assigned to the object and a code allowing package subprograms to check that the variable has not been modified by an illegal operations. Indeed, the only legal kind of operation in which such a pseudo-variable may appear in an application program is parameter transmission. Any other use (e.g., integer addition) is forbidden and will normally be detected in the next call to a subprogram of the package.

This technique seems the best way of adapting abstract data type concept to Fortran: an object is only available through its name and a set of well-delimited operations. The resulting programming style is not, of course, typical of Fortran. In the box below, an example of Gescran programming appears.

5. Conclusion

We believe that the principles expounded upon in this paper may be applied with equal success to widely different kinds of software, and we

```
INTEGER SCREE, WINDO1, WINDO2          Declare pseudo-variables.
.....
CALL DEFSGE (SCREE)                    Pseudo-declaration of SCREE as a
.....                                    screen pseudo-variable.
CALL CREWGE (WINDO1, SCREE, 2, 5, 7, 12) Pseudo-declaration of WINDO1 and
CALL CREWGE (WINDO2, SCREE, 6, 15, 1, 4)   WINDO2 as windows in screen
                                           SCREE.
.....
CALL REPWGE (WINDO1, string 1)         Initialize contents of windows (RE-
CALL REPWGE (WINDO2, string 2)           Place contents of windows).
.....
CALL BRIWGE (WINDO1, 'B')              Define WINDO1 as bright (BRIlliance
.....                                    of Window).
CALL WRISGE (SCREE)                    Display SCREE.
```

hope that our discussion has shed some light on a key problem in software engineering: how to write reusable software. It should be pointed out, however, that all of our products are conceptually small. This was a deliberate decision on our part since we felt modest-sized team best succeeds with simple, efficient, and reliable programs, rather than large-scale, ambitious ones. Although we feel many of our methods would apply successfully to larger projects, we do recognize that their applicability to, say, a vast numerical library remains to be proved.

Acknowledgments

The work reported on here involved, in particular, E. Audin, G. Brisson, E. de Drouas, and B. Logez. Many others provided advice—most notably, A. Bossavit. At a meeting of the groupe "Génie Logiciel" (Software Enginnering) of AFCET-TTI (French Computer Society), additional useful suggestions were made. The author is also indebted to I. Qualters for many improvements in the style of this paper, and to the reviewers for their comments.

References

1. Dahl, O.J., Myrhaug, B., and Nygaard, K. *Simula 67: Common Base Language.* Rep. S-10, Norsk Regnesentral, Oslo, Norway, 1970. The original description of the first of the modern modular languages. Assumes the Algol 60 report as a prerequisite; an integrated report is currently in preparation.

2. Honeywell, Inc. *The Ada Programming Language—Proposed Standard Document.* U.S. Dept. of Defense, 1980, Washington, D.C. The report on the new U.S. Department of Defense language, designed by a team led by J. Ichbiah.

3. Jackson, M.A. *Principles of Program Design.* Academic Press, London, 1975. Describes a popular program design methodology, based on the idea that a program's structure should be modeled on the structure of the data it manipulates. Prime target: business data processing.

4. Kernighan, B.W., and Plauger, P.J. *Software Tools.* Addison-Wesley, Reading, Mass., 1976. A methodology for constructing composable programs, with a bottom-up presentation of a number of examples.

5. Kernighan, B.W. Ratfor—A preprocessor for rational Fortran. *Software—Practice and Experience* (Oct. 1975). One of the most popular Fortran preprocessors.

6. Meyer, B., and Baudoin, C. *Méthodes de Programmation.* Eyrolles, Paris, 1978. A fairly comprehensive survey on programming methodology, programming techniques, basic algorithms, and data structures.

7. Meyer, B. *Un Ramasse-Miettes par Tri.* Rep. Atelier Logiciel 8, EDF—Direction des Etudes et Recherches, Sept. 1978. Describes a particular garbage collection algorithm used in a package for text manipulation.

8. Meyer, B. Sur quelques concepts modernes des langages de programmation et leur Representation en Simula 67. AFCET-GROPLAN, Vol. 9, Cargèse, 1979, pp. 331–395. How Simula supports modern programming concepts, such as modularity, genericity, top-down design of both algorithms and data structures, etc.

9. Norsk Regnesentral. *Simula 67 for IBM System/360—User's Guide; Simula 67 for IBM System 360—Programmer's Guide.* Pub. S-24-1 and S-23-1, Oslo, Norway, 1975. Reference for the IBM Simula implementation, a programming environment with desirable features like separate compilation and symbolic debugging.

10. Rose, L.R., and Hellerman, H. Portable character processing in Fortran and fixed character environments. *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 76), 176–185. A package for text manipulation.

11. Saltel, E. *Manuel Fortran 30.* IRIA, Rocquencourt, France, 1978. An extension of Fortran which allows graphic processing.

12. Wirth, N. Program development by stepwise refinement. *Comm. ACM 14*, 4 (April 1971), 221–227. A classic reference on the top-down design of programs. Mentions that the refinement process should apply to data structures as well as the algorithmic part.

13. Woodfield, S.W., Dunmore, H.E., and Shen, V.Y. The effect of modularization and coments on program comprehension. Proc. 5th Internat. Conf. Software Eng., San Diego, Calif., March 1981, pp. 215–223. An experimental study on what factors affect the readability of programs. Some results support the view that abstract data types are a good basis on which to construct modules.