MODULA-2 AND SOFTWARE ENGINEERING

INFORMATICA 1/90

Gustav Pomberger
INstitut für Wirtschaftsinformatik
Johannes Kepler Universitl of Linz
A-4040 Linz, Austria, e-mail: K2G0190 a AEARN

Keywords: modula-2, software engineering, models, principles

Presented at the 1st Int'l Modula-2 Conference, October 12-13, 1989 Bled-Yugoslavia

OVERVIEW

The title of this paper draws together a programming language, Modula–2, and a discipline in the area of computer science, software engineering. This raises several questions:

What is meant by software engineering? This is by no means intended to be a rhetorical question, for the perceptions of software engineering's tasks varies pronouncedly from the viewpoints of theoreticians and practitioners.

Even greater divergence can be found on the question of what a programming language has to do with software engineering. Some consider the choice of a programming language of utmost importance to the success of a software project and the quality of the resulting product, while others view the language as the least important tool of the development process.

Even the question of whether programming is more a science or more an art (or perhaps even a craft) evokes avid disagreement. I do not want to renew this old feud; I simply want to establish that elements of all of them are inherent in software development at this time, and this is likely to remain the case in the future.

When I use the term software engineering, I mean the application of scientific knowledge for the efficient production and application of reliable and efficient software (see [16]).

The successful development of large software systems is usually a multistage process. It usually begins with the determination and documentation of the functions and individual actions that are expected of the software system. This leads in the specification phase to a contract between the client and the software developer (requirements definition) that precisely delineates what the software system must be capable of.

The specification phase is followed by the design phase, which determines what kind of system architecture can meet the given requirements. The implementation phase attends to the realization of the complete design concept in a programming language.

The implementation of every single system component must be systematically tested. Subsequently the whole system must be tested with the goal of finding as many errors as possible and assuring that the implementation meets the requirements definition.

Upon completion of the test phase the software is installed and handed over to the client. The task of software maintenance is both to correct errors that arise during operation and to make system modifications and extensions. This task again includes all activities mentioned above—from the revision of the requirements analysis through renewed testing.

An engineering discipline is characterized by the construction of tools that help to systemize and rationalize the product development process, to improve the quality of products, and to guarantee efficient maintenance. A particularly important step in this direction was the development of programming languages as tools intended to help to achieve these goals. Unlike many others, I agree with B.W. Boehm that "choosing a programming language is like choosing a wife. It is hard to undo after getting involved and not to be taken lightly."

I likewise agree with R. Wiener and R. Sincovec [22] that "the choice of a programming language for implementing a large-scale software system is critical because the features of a programming language are strongly related to the software engineering process. Languages differ in the degree to which they support: readability, modular software construction, the control of side effects, information hiding, data abstraction, structured flow control, separate compilation with consistency checking, type checking among various components, dynamic memory management, and run-time checking. Languages that offer strong support in the above-listed areas provide the basis for constructing reliable and maintainable software."

I will discuss to what extent the principles of software engineering known today are supported (or not supported) by Modula-2. Detail is restricted by the size and extent of this paper. For this reason only the principles that I consider most important will be discussed. I will briefly discuss the characteristics of software development models in order to be able to explore:

- specification principles
  - requirement exploration by prototyping
- design principles
  - module-oriented architecture design
  - abstract data structures
  - abstract data types
  - functional abstraction
  - object-oriented design principles
- other evaluation criteria
  - division of labor in software development
  - structuring in the small (structured programming)
  - guaranteeing reliability and maintainability
  - exception handling
  - reusability of library modules
  - portability

SOFTWARE DEVELOPMENT MODELS

Wherever people are confronted with complex design tasks to be solved, they attempt to systematically organize the problem solving process, that is, to define an approach model. Such a model determines which criteria are to govern the problem solving process. It decomposes the problem-solving process into managable steps and determines what results must be produced after execution of a given step. This enables a stepwise planning, decision and implementation process.

These steps collectively and the chronological order of their execution is known as the software life cycle, an already classical term in computer science. The software life cycle has been described in numerous variations and forms (see [8], [20], [16], and [18]).

Studies have shown that the life cycle-oriented development method is the most commonly used approach in current software development, and that it has in general paid off. Application in the field, however, has also shown the limits and the weaknesses of this approach:

The model is based upon the (incorrect) assumption that the development process tends to be linear and that iterations between phases occur only as exceptions to the rule. Strict application of this development method requires that one phase can only be begun after the preceding phase is completed, that is, when the respective intermediate products are available. In reality, however, a complete specification or a suitable system architecture can seldom be produced straightoff. Usually the later phases have a strong impact on the earlier phases.

The strict discrimination of the individual phases is an unacceptable idealization. In reality the activities of the phases overlap and interaction between phases is much more complex than that exhibited in the sequential input/output model.

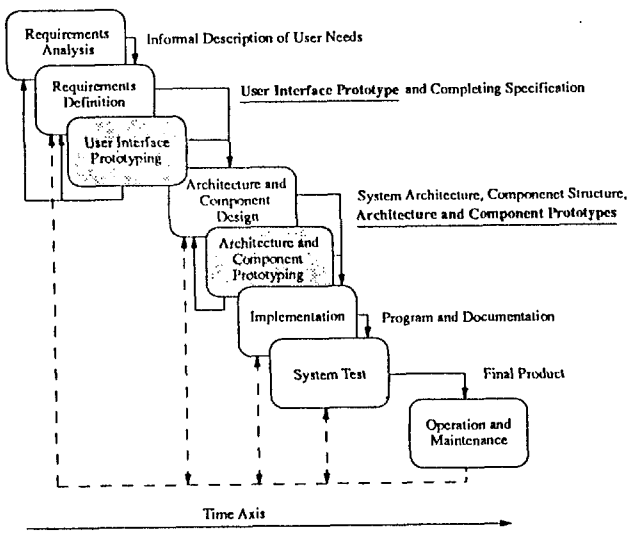

Figure 1 Prototyping-Orineted Software Life Cycle

The strictly sequential approach leads to tangible products or components being available only at a relatively late stage. Yet experience shows that the validation process cannot get by without experiments close to reality. Furthermore, modifications requested by the client can only be expressed relatively late, and integrating them at that stage can lead to substantial overhead.

It is often assumed—and current reports from research and industry confirm this assumption—that a prototyping-oriented development methodology can resolve some of the weaknesses of the life cycle-oriented development approach. A prototyping-oriented development is not radically different from a purely phase-oriented development strategy. Furthermore, the two are to be viewed more as complementary than as alternative. They differ most in the procedures and the results produced in the individual phases. Although the distinction of phases is maintained, problem analysis and specification overlap chronologically a great deal, and design, implementation and testing very much blend into one another (see Figure 1).

SPECIFICATION PRINCIPLES—EXPLORATION OF USER REQUIREMENTS

As our development model shows, one element of knowledge inherent in our definition of software engineering is that the specification and design processes should be carried out in a prototyping-supported manner.

The development of the user interface, for example, proves an exceptionally difficult task because the evaluation thereof is guided by highly subjective criteria and the user is hardly able to define in advance what he/she considers to be convenient interaction. Prototyping is an important—and, from my point of view, in most cases absolutely necessary—vehicle for the exploration of user requirements and thus for the specification of user interfaces.

We normally distinguish two approaches to prototyping: *reusable code* and *executable specifications*. Modern programming languages like Modula-2 are significantly better suited for producing reusable components than was the case in older programming languages. Modula-2 is particularly handy for the building of module libraries. From the viewpoint of prototyping, however, a number of problems remain unsolved if one uses conventional programming languages such as Modula-2 for prototyping activities:

- How can the functionality of a library modules be provided generally enough that they can be integrated into a given prototype?
- The degree of abstraction of Modula-2 modules is too low; a prototype designer must revise code for every modification, no matter how small, and make his/her changes directly in the code; details of the prototype cannot be discussed with the user.
- Turnaround times for iterative refinements in a prototype are simply too high.

Although the availability of module libraries is steadily improving and the taxonomy of software components is beginning to emerge (already it is possible to distinguish components such as mathematical routine packages, message channels, input/output packages, parsers, scanners and filters to name a few), Modula-2 libraries are only to a limited extent (if at all) capable of meeting the demands of reusability of code as required for prototyping.

The other approach to rapid prototyping, executable specifications (an object of intense research efforts) is likewise not supported by Modula-2.

Since on the one hand we use Modula-2 as our implementation language in most cases in our research group (and the choice of Modula-2 is to be credited with considerable increases in efficiency and quality), and on the other hand we have recognized the value of prototyping-oriented software development and evaluated this in several research projects, it became necessary to develop special tools for prototyping.

For the prototyping process during the analysis and specification phases, we developed a declarative language for the description of executable specifications—our User Interface Specification Language (UISL, see [12] and [17]). Searching for methods for integration of high level prototypes and application parts written in Modula-2 as well as for validation of a system architecture before completely implementing it, we developed SCT, a tool for hybrid execution of hybrid software systems (see [1] and [2]). It allows for hybrid execution of Modula-2 software systems at any time during their development. Designed but not implemented modules are simulated, partially coded modules are interpreted, and modules which are coded and tested are directly executed. Furthermore, SCT allows for execution of hybrid software systems (systems written in different languages). This is achieved by providing the possibility of adding new execution tools to SCT's hybrid execution system (e.g., an interpreter for a user interface description language).

Applying SCT high level prototypes can be easily enhanced with Modula-2 code, allowing the development of better exploratory and evolutionary prototypes. Furthermore, SCT supports the validation of system architectures represented by Modula-2 definition modules by simulating data and control flows. Finally, SCT provides a comfortable interpretative programming environment allowing for fast implementation and experimentation with different realizations of the functionality provided by a module.

SOFTWARE DESIGN PRINCIPLES

The task of the design phase is the determination of the architecture of a software system—that is, to decide how to build the proposed system—with the goal of achieving an implementation that is as efficient as possible and meets all quality requirements. Because of the practically unlimited number of possibilities of determining the design of a planned system, the decisions made and the methods used in this phase pronouncedly influence the quality of the product and thereby its maintenance costs and degree of reliability.

The production of complex program systems necessitates a division of labor; that is, multiple persons are involved in the software development. It is clear that software development is a creative process, that the experience, creativity and innovation of the designer significantly affects the quality of the product. But as a rule the complexity of design decisions is so high that a systematic approach—a method and associated design principles—must be adhered to in order to guarantee a resulting product that is reliable and easy to maintain.

All software design involves a process of abstraction. Objects and operations identified in the real world domain must be modelled and expressed as corresponding operations and objects of the problem-solving domain.

Module-oriented and *object-oriented software design* are fundamental design principles resulting from computer science research in the 60s and 70s. Wiener and Sincovec write [22]:

"No longer is it necessary for the system designer to map the problem domain to the predefined data and control structure present in the implementation language. Instead, the designer may create his or her own abstract data types and functional abstractions and map the real world domain to these programmer-created abstractions. The mapping, incidentally, may be much more natural because of the virtually unlimited range of abstract types that can be invented by the software designer. ... the payoff for modular software design and implementation occurs when repairs or additions must be made to a software system."

Module-Oriented Design Principles

The goal of modular system design is the decomposition of a program system into a hierarchy of abstractions about which Wirth writes [24]: "The principle motivation behind the partitioning of a program into modules is—beside the use of modules provided by other programmers—the establishment of a hierarchy of abstraction."

The pillars of modular system architecture are *module independence* and *data abstraction*. Module independence (freedom from interference) means that any module can be replaced by another module that adheres to the module interface without necessitating further changes in the system. That is, it must be possible to change details of the implementation of a particular module without influencing the remaining system components.

The basic building blocks of modularly constructed software systems are:

- abstract data structures
- abstract data types
- functional abstraction
- abstract, explicitly defined module interfaces

Although software engineering courses often teach that design should occur completely independently of the implementation language. I believe that this is only useful if the implementation language does not meet the requirements of software engineering. We are aware that a language reflects the habits and thought patterns of its designer. There is even a relationship between a natural language and the way a person who speaks the language thinks. The same is true for programming languages. The knowledge that lent it its structure and the concepts that form its basis influenced the way a programmer thinks, his/her design style, and the structure of the system he/she designs. The choice of a programming language often even determines how the task is solved because the language supports or excludes certain approaches to a solution. For example, a recursive tree traversal would never come to a Fortan programmer's mind.

Furthermore, we expect a good implementation language to be able to reflect the decomposition structures, abstraction levels, data structures and module interfaces that are identified in the design stage and that these can be tested at the interface level before all the implementation details are known.

The degree to which these requirements can be met is dependent upon the choice of a programming language. The question is to what extent Modula-2 supports the above-named criteria for modular system architectures.

The Modula-2 Module Concept

The realization of the module definition as given above is supported in an elegant manner by the module concept of Modula-2. The modular structure of Modula-2 can be viewed as a fence that encloses objects (data structures and procedures) and encapsules them apart from their environment. This fence can be opened for the purpose of communicating with the environment. However, the programmer must explicitly establish which objects are to be made known (that is, exported) to the outside and which objects the

module will need (that is, import) from its environment. This meets the requirements of explicitly defined module interfaces.

From the viewpoint of the abstraction principle, the export interface can be seen as its specification. It contains all information regarding what the module is expected to do (that is, what objects and functions it makes available) and hides all details of the implementation thereof. It is thus also useful to separate the texts of the module specification and its implementation description. Modula-2 meets this requirement by separating the definition and implementation parts of a module.

One of the most important aspects of modularly constructed software systems is thus an explicit description of mutual effects (that is, interdependencies) among modules. The importance of such explicitness follows from the observation that all the effects of a local change on the global system must be completely determined by the dependency relations. In Modula-2 interfaces of modules as seen by the programmers are called definitions. Such module definitions may be regarded as public projections, and there is exactly one public projection of each module.

But this situation is less than satisfactory. In practice we often encounter situtions in which multiple views of a module can be seen as befitting the problem. Consider, for example, a module X for managing assembly lists in a production planning and control system. It is clear that a module A from the area of design requires different access functions than a module B from the area of work scheduling or a module C from the area of material disposition. A, B and C all work with the encapsuled data structures in X, although in different ways and with different requirements for access to the data structures of the assembly list encapsulated in X.

This is just one of many examples in which multiple interfaces to a single module are necessary, each with different levels of abstraction, in order to guarantee adequate application of the module with respect to the problem at hand.

Due to the one-to-one correspondence of modules and interface descriptions in Modula-2, multiple interfaces cannot be satisfactorily realized. Either all the different views are packed into a single interface—which increases the complexity of the interface, reduces the safety of the module, and destroys part of the abstraction—or the implementation is duplicated—that is, reusability is lost and maintainability is reduced.

Multiple interfaces of modules are thus an important concept in software engineering that is not supported by Modula-2. Ideas on the implementation of multiple interfaces can be found in [10].

Abstract Data Structures (Information Hiding)

The basic concept of Modula-2 is the establishment of a hierarchy of abstractions. Naturally, this includes the implementation of *abstract data structures*. The problem of specially identifying access operations to a(n abstract) data structure is solved in Modula-2 by dividing a capsule into two parts: one part visible to the user (the specifications or interface part) and containing the declaration of all access operations and any exported data types; the other part invisible to the user (the implementation part) and containing declarations of encapsulated data and algorithms in the capsule.

The module concept of Modula-2 includes the export of not just procedures, data types and constants; variables can likewise be exported (for example, to make access to a single data element more efficient). If a variable is exported, its value can be changed by the importing module. However, this violates the principle of information hiding and it must be clear to the importer that he/she is working with global data, and the efficiency thereby attained is countered by the disadvantages of exchanging data via global variables.

Modula-2 thus permits the implementation of data capsules, although the principle of information hiding is incompletely realized due to the possibility of exporting inner data structures together with their structure. In this sense it would be desirable to have exported variables that can be read but not written to by the client.

Abstract Data Types

Abstract data types are necessary when multiple examples of an abstract data structure are to be defined. Abstract data types can be implemented in Modula-2 by means of the module concept combined with the concept of opaque data types.

An abstract data type is defined as an opaque type in the definition module; that is, its realization remains hidden from the user and is determined in the implementation module—in contrast to Ada—which is a considerable advantage from the viewpoint of software engineering.

Unfortunately there is a catch to using opaque data types in Modula-2. Since the storage requirements of abstract data types must be known when the definition module is compiled, Modula-2 requires that the concrete type assigned to an abstract type must be of fixed length—that is, it must be a pointer type. Other types, in particular ARRAYs and RECORDs, are not permitted as abstract data types. They can, however, be realized as dynamically created objects and their pointer can be viewed as an abstract data type. This means a slight loss of efficiency, however. I personally consider the advantage of abstract data types to be greater than the disadvantage of the loss of efficiency.

It is much worse to have to dynamically allocate variables of abstract data types and to have to explicitly free their storage. Furthermore, the statement x:=y does not store a copy of y in x. This is a dangerous pitfall that can cause less experienced Modula-2 programmers to avoid the use of abstract data types.

In the process of designing a software system, we usually encounter modules or procedures that have a similar purpose but operate on data objects of different types, for example, modules for stacks, queues, trees, etc. What we want to have is a construct that permits the definition of templates for program units that need to be written only once and then tailored to the particular needs at translation time. This would be possible with *generic units*, but generic units are not available in Modula-2.

The data type WORD or ARRAY OF WORD serves as a lifebuoy in such cases. This allows, for example, the creation of a very general stack suitable for accepting simple objects (for example, CARDINAL) as well as structured objects (such as ARRAYs and RECORDs). I consider the omission of generic units (which are most uncomfortable from the viewpoint of the compiler designer) to be a clever decision which, because of the self-help available in the type WORD, is also acceptable at the practitioner level.

Functional Abstraction

Many software developmenters construct the software system architecture as a hierarchy of functional components, i.e., they employ the method of task-oriented stepwise refinement. Functional aspects are the focus of the method. Starting with the functional requirements, the task is decomposed into subtasks; each subtask (functional component) is then handled separately and again decomposed into subtasks until the resulting subtasks become so simple that they can be described with algorithms. That is, top down design proceeds from the general to the specific, from an identification of major system components to subcomponents and sub-subcomponents and so forth.

In the implementation we want to realize the hierarchical levels of the system architecture by mapping the functional components onto a set of (possibly nested) procedures that are used to implement the functional abstractions. The only language features we need to support top down design by stepwise refinement are procedures and the ability to group functional components into functional subsystems.

Through its procedure and module concepts, Modula-2 completely supports this method and permits the interfaces of the functional components and functional subsystems to be described precisely, yet, as the design process requires, abstractly enough.

Object-Oriented Design Principles

A design principle which has aroused a great deal of interest recently in computer science is *object-oriented system design*. Reduced to its fundamentals, object-oriented programming generates software by reproducing object descriptions. An object description contains definitions of data along with the specifications of actions that can be applied to these data.

In contrast to modular programming, object descriptions are only a kind of type description and do not form actually existing constructs as does a module in the sense of Modula-2. Only when an object description is instantiated is an object created.

However, object-oriented programming is more than just using abstract data types. It also involves inheritance and dynamic binding.

An important property of object-oriented system design is that the object descriptions do not contain complete definitions of the object's behavior and attributes, that is, all its data and actions. The object descriptions are ordered in a hierarchy in such a way that at any given hierarchy level only such data and actions are specified as were not already defined in superordinate object descriptions. Modifications of data and actions are thereby made without altering the superordinate object descriptions. This distinguishes object-oriented software development from module-oriented programming, in which the reuse of a module is only possible without changes in its implementation if the module's function completely fits into the new context without change.

The strength of object-oriented system design lies in the possibility of incrementally enhancing and adapting object descriptions without touching their code in the process. Instead of the libraries used in modular programming—with their reusable function modules whose components can be used in the construction of software—object-oriented programming uses libraries of object description hierarchies that form application frameworks. Examples include Smalltalk [9], MacApp [19] and ET++ [21].

As a rule, object-oriented programming builds on applications or parts of applications that are adapted to specific requirements, yet without changing these parts themselves. Thus later modifications can be made on the prefabricated application parts that remain completely transparent and spread to all derived applications without any further overhead.

The requirements placed on programming languages which support object-oriented system construction match those for languages which support modular system construction in many respects. In addition, they must support the following concepts (see [3]):

- *Data abstraction:* The description of abstract data types in the sense that they can occur directly in the declaration of other data structures must be possible.
- *Inheritance:* It must be possible to derive new data types by extending or modifying attributes and operations of existing types without needing to modify the description of the base types. Instances of a class C built on the basis of a class B are said to inherit the properties of B.
- *Polymorphism:* The compatibility of derived data types and their base types must be guaranteed. Object variables must be able to assume values of different (but related) data types at run-time.
- *Dynamic binding:* In the course of operations with objects, there must be the possibility of determining at run-time the concrete actions to be executed (dependent on the current dynamic data type of the objects).

In the object-oriented nomenclature, an abstract data type is known as a class. Every class defines which attributes its instances (the so-called objects) have and which operations are possible with them. Activating an operation with an object is often termed sending a message to the object. The object reacts by executing a method. A method describes which actions are to serve as the realization of an operation. This assignment of methods to messages is determined for each class by the respective class definition. The effect of sending a message differs from procedure invocations in conventional programming languages in that the determination of which method is to be executed occurs at run time.

The question is whether Modula-2 can be used to realize object-oriented system architectures and, if so, how it can be done. Object-oriented programming does not necessarily require an object-oriented programming language. Suggestions on how to implement objects in Modula-2 can be found in [4]. Every individual class can be defined in a separate definition module. Objects can be defined as pointers to records with two components: a pointer to a data structure describing its class and a pointer to the object's data (that is, instance variables). A class can be defined by a record containing a pointer to its superclass, the name of the class, and a collection of procedure variables which represent the messages understood by objects of this class.

Of course, some deficiencies must also be mentioned (see [4]). The programmer must be aware of the fact that objects are implemented as pointers. Thus, each object must explicitly be created. Also, the statement x:=y does not create a copy of the object y. Instead, a message send must be used. Changing a superclass' definition module requires changes to all of its subclasses. The requirement that every

class must be defined in a separate definition module can lead to a large collection of modules that is difficult to understand and maintain.

Thus I cannot recommend Modula-2 to construct object-oriented system architectures as it was not designed as an object-oriented language. But many of the deficiencies mentioned above can be removed by attaching minor extensions to Modula-2.

There are, of course, some object-oriented extensions of Modula-2, among them Modula-3 [5] and p1 Modula [11]. Niklaus Wirth himself also designed a new language named Oberon [23] that is based on Modula-2. Oberon was not designed as an object-oriented language either, but readily lends itself to the concept using type extensions and procedure variables. And an experimental extension of Oberon, called Object Oberon, has been developed that incorporates the concepts of class, method and message [14]. Including these concepts in Oberon improves its capabilities for object-oriented programming.

OTHER EVALUATION CRITERIA

I have discussed to what extent Modula-2 supports the most important software engineering principles for the exploration of user requirements (prototyping), for mastering complexity (structuring in the large), for engineering interfaces (information hiding, data abstraction), and for the design of the architecture of software systems (modular system construction, object-oriented system construction).

Beyond these aspects, we are interested from a software engineering viewpoint in several other criteria and how these are supported by the choice of Modula-2 as implementation language, for example:

- division of labor in software development
- structuring in the small (structured programming)
- guaranteeing reliability and maintainability
- exception handling
- reusability of library modules

Division of Labor in Software Development

The process of dividing the work load in software development is significantly supported if:

- separate interface description and implementation description of the system components is possible;
- separate compilation of units with strict cross checking is possible;
- type consistency checking between various components is provided; and
- the execution of a program unit is automatically prevented if the interface of a user component was modified and no consistency check followed the modification.

All of these properties are supported by Modula-2. This reduces the chances of the hard-to-localize kind of errors that arise from incompatible interfaces in divided-labor software development.

The concept of separate compilation coupled with the concept of strict type binding contributes to drastically increased productivity in a divided labor setting in a revolutionary way that is unfathomable to programmers in conventional languages such as Fortran or Cobol, while simultaneously (and at almost no additional cost) heightening reliability and maintainability.

Structuring in the Small

The goal of structuring the control flow of algorithms is to establish a correspondence between the static formulation of an algorithm and its dynamic behavior, to thereby reduce its susceptability to errors, and to enable the verification of the algorithm. The most important measure in this direction is the avoidance of unlimited flow structures which result from undisciplined use of unconditional transfers of control (goto statements). Thus many consider the absolute avoidance of such statements to be a fundamental requirement of structured programming, and they insist that control flow is to be structured by including only constructs that have a single entry and a single exit.

Modula-2 does not completely meet all these requirements of fundamentalist structured programming, for Modula-2 provides the RETURN and EXIT statements. But these disguised gotos do not compromise the software engineering principle in an essential manner, and they sometimes increase the efficiency and readability of programs if a loop/exit is used instead of some boolean variables and a conditional transfer test to circumvent the need for a loop exit statement. This latter technique often detracts from program clarity.

Although it is, of course, clear that good programming style is not characterized by the absence of goto statements alone, the lack of a goto statement in Modula-2 forces programming with well-defined transfers of control. This is an important property of Modula-2 from the viewpoint of software engineering, and I agree with B. Meyer's observation [13]:

"It is hard to understand that, twenty years after 1968, a single letter about the goto construction should trigger endless letters to the Communications of the ACM, many of them advocating the use of gotos. Why not Roman numerals?"

Guaranteeing Reliability and Maintainability

Prerequisites to a *reliable, maintainable* software product include clear, consistent specifications, followed by the clean design of a modular architecture, followed by a readable description of the implementation, and culminated by a rigorous, systematic testing procedure aimed at both the individual components and their interaction.

Module independence is certainly one of the most important factors in the design of reliable and maintainable software systems. Guaranteeing the reliability and maintainability of a program system is less expensive as the components of a program system are easier to tune, to correct or to adapt to new requirements without affecting other parts of the system. The ability of the software designers to create module independence is very much related to the choice of the programming language to be used in implementing the system. The prominent concepts of Modula-2, such as information hiding, data abstraction, splitting definition and implementation of modules, and side effect avoidance through the use of proper variable scoping greatly affect the type of design and implementation and enhance reliability and maintainability considerably.

The *readability* of an implementation description is likewise an important criterion for maintainability. It dependends on the structuredness of the system, on programming

style, and on the expressive power of the implementation language used. Significant improvements in program readability result from:

- the use of descriptive names of arbitrary length
- the ability to define type names
- the ability to use abstract data types
- the compulsion to use formal object declarations (this serves as an identifier glossary)
- the possibility of reusing identifier names in the same program at different levels of locality

The clear lexical and syntactic construction of Modula-2 and the possibility of meeting the criteria named above assure a high documentation value. So long as an appropriate programming style is maintained, Modula-2 programs are more readable than PL/I, Cobol, Fortran or (in particular) C programs.

An additional important criterion for reliability and maintainability of a software system is its *testability*, which means its suitability to checking its correctness and localizing errors. The most important criteria for testability, most of which are fulfilled by Modula-2 are:

+ *Modularity of the system:* The system architecture is formed by a hierarchy of abstractions (modules). The interaction of modules is explicitly defined (import/export interfaces). Constructs are provided for structuring modules (functional components). Each functional component of a module has its own scope (nested locality).

+ *The ability to avoid side effects:* Communication among program units can only occur via explicitly described interfaces. Each program component has its own scope. Combination of data objects is only possible if their data types are compatible; implicit conversions are precluded.

+ *The ability to guarantee information hiding and data abstraction:* The data contents of a module and their representations are not visible to the outside and are thus protected from procedures that access them. Only procedures declared locally to the data can access the data structures, that is, know their concrete representations. The use of external data structures is precluded (module decoupling).

+ *Structuring of the control flow:* The control flow of an algorithm reflects its static structure. That is, the exclusive use of flow structure constructs with a single entry and a single exit is encouraged.

+ *The ability to check the consistency of module and procedure interfaces:* Interface descriptions (import/export procedure interfaces) are of a nature that a compile time check can be made to determine whether the client and the server (module/procedure) match one another.

+ *The readability of the implementation* (see above).

+ *The availability of run time checking facilities* such as range check, index check, etc.

– *The ability to specify semantic aspects (assertion mechanism):* Procedures and lower level units (i.e., loops) can be provided with assertions that describe semantic aspects of the program segment and can be evaluated at run time. The underlying idea (see [13])is programming by contract: "Every structure is charged with a precise task, defined by a specification that states precisely the obligations on the client, limiting the routine's responsibility (the preconditions) and the obligations on the routine, guaranteeing the client a certain result (the postconditions)."

Modula-2 not only permits but considerably supports these criteria for increased testability and thereby for heightening the reliability and maintainability with a single exception. An *assertion mechanism* as it is found, e.g., in Eiffel [13] is lacking in Modula-2. This drawback is, however, easier to accept in modular programming than in object-oriented programming because dynamic binding can obscure what actually happens in an object-oriented program.

Modula-2 supports measures to guarantee reliability and maintainability to an incomparably greater extent than Fortran, Cobol and C, the most-used programming languages today. Studies in our research area (development of software engineering tools) have shown that the overhead for testing and maintaining of projects with Modula-2 as implementation language were less than 50% of the overhead in similar projects in which PL/I and C were used.

Exception Handling

In the execution of a program, events or conditions can occur (e.g., protocol errors in the transmission of data) that require special treatment. Language constructs for describing and handling such events (exceptions) contribute to the reliability and clarity of program systems. Thus a number of programming languages (e.g., Ada, Clu, Eiffel) incorporate language constructs for exception handling.

Such constructs do not exist in Modula-2, an absence which has often been identified as a drawback of the language. I cannot agree with this verdict. In all our projects I never encountered a case where programming out exception handling posed difficulties or detracted from the clarity of the program. Furthermore, it is easily possible to implement a mechanism for exception handling in Modula-2 with the help of coroutines and/or library modules.

Reusability of Library Modules

Modula-2 has provided us in particular with the separation of an interface description from the implementation of a module and the possibility of modifying the implementation without needing to change anything else in the rest of the system in which the module is imbedded (not even recompilation). Since the introduction of the module construct in programming languages, programmers expect significant improvement in the reusability of prefabricated software units as well as the creation and distribution of powerful module libraries.

Typical library modules contain a collection of procedures that implement often needed functions and belong together in some manner, e.g., a trigonometry module; or they implement an abstract data type that provides the client with a new data type and the operations defined on it, e.g., stack, queue, tree, sparse matrices; or they model physical systems to operate between hardware components and the rest of the software system, e.g., device drivers, communications modules. In addition to physical systems, logical/conceptional systems are naturally likewise modelled, i.e., made useful for other software components at a higher abstraction level, e.g., graphic modules, database modules.

In software engineering practice the situation often arises that a library module almost but not quite meets the

requirements of a new application. Modifications become necessary. If changes only affect the implementation part, there is less problem. However, the definition part is often affected as well (perhaps a new albeit trivial operation is needed). A change in the definition part carries with it the ramifications that all client systems have to be compiled anew. In order to avoid this, there is no alternative but to copy the original module and to make the changes in the copy. With time this can lead to a whole family of different yet closely related modules. If a fundamental aspect of this module family needs to be modified, an aspect which is common to all the members, then each member of the module family has to be modified.

Another drawback that restricts reusability is that modules in the sense of Modula-2 define a static object and do not permit the definition of an object type. A module can thus not be defined once and be repeatedly instantiated. This proves to be a particular impediment in modelling abstract data types. I have discussed how abstract data types are reproduced with the module concept: A data object must be explicitly allocated with the invocation of a procedure; the data type itself is referenced with an opaque pointer. Thus reproduced abstract data types cannot occur directly in other data structures, or be transferred to other processes, or be output directly to files; one has only the opaque pointer as reference to the abstract data type. Special procedures have to be defined for such operations for each data object and have to be invoked by the client at the right time. A disagreeable side effect is that the client has to treat abstract data types differently from real data types.

In order to guarantee a sufficient measure of reusabililty, it must be possible to apply abstract data types for defining arbitrary data structures. Furthermore, it must be possible to enhance abstract data types in a simple and flexible manner without violating the principle of information hiding. This is not possible in Modula-2—or at least only in a very troublesome manner. (I alluded to this in the section on object-oriented system construction.) The reusability of library modules in Modula-2 thus does not completely meet the requirements of modern software engineering. Thus in our software development environment module libraries were used only for elementary tasks.

SUMMARY

My goal was to subject Modula-2 to critical analysis. I did not do this on the level of D. Moffat, who wrote [15]: "Modula-2 is not a general purpose language. Every general purpose language must also include some way to deal with large fixed-precision numbers for monetary quantities." I also did not seek to discuss what N. Wirth's Modula-2 Report did not precisely define, as, e.g., in B.J. Cornelius [6], [7]. Instead I sought to give an overview of the extent to which the language meets the requirements of software engineering at the end of the 80s.

Needless to say, at the start of this decade Modula-2 was a jewel—indeed, a diamond—that enriched the programming landscape. The ability to combine multiple procedures into a module, information hiding, separate compilation with full interface consistency checking, the ability to formulate parallel processes by means of the elementary concept of coroutines with various synchronization mechanisms, the support of most of the concepts of software engineering familiar at that time, the high documentation value of Modula-2, the compactness of the language, and the elegant syntax compared to other programming languages made Modula-2 a powerful tool for software engineers. All this makes it most incomprehensible that only a small segment of software engineering, mainly the academic sector, made use of this milestone language.

Today, at the end of the 80s, the world looks a little different. Software engineering has continued to develop—inspired by the fruitful works of N. Wirth and others. New programming paradigms have their consolidation phases behind them and new requirements for programming languages have evolved as a consequence. From my point of view the most important are: the availability of constructs for realizing object-oriented software architectures, the ability to create multiple interfaces to modules with respect to objects, and assertion mechanisms provided by a language to increase the reliability of programs. It is clear that the programming languages of the 70s to which Modula-2 belongs cannot completely meet these requirements. But from my point of view, these enhancements can be attached to Modula-2 with minor extensions, the subject of work currently in progress.

One step in this direction was, as mentioned above, the development of Oberon and the enhancements that led to Object Oberon. Modula-2, in terms of the fundamental concepts of the language and its cleanness and simplicity, forms a significantly better basis for further development in the directions mentioned than other programming languages, in particular C, which is so questionable from a software engineering viewpoint.

Our research group is among those that are working on further developments in the area of programming languages—naturally based on the solid foundation that Modula-2 provides.

REFERENCES

1. Bischofberger W., Keller R., 1989, Enhancing the Software Life Cycle by Prototyping, Structured Programming, Vol. 10, No. 1, Springer.

2. Bischofberger W., Pomberger G., 1989, SCT—A Tool for Hybrid Execution of Hybrid Software Systems, Proceedings of the First Annual Modula-2 Conference, Bled, Yugoslavia.

3. Blaschek G., Pomberger G., Stritzinger A., 1989, A Comparison of Object-Oriented Programming Languages, Structured Programming, Vol. 10, No. 4, Springer.

4. Blaschek G., 1989, Implementation of Objects in Modula-2, Structured Programming, Vol. 10, No. 3, Springer.

5. Cardelli L. et al., 1988, Modula-3 Report, Olivetti Research Center.

6. Cornelius B.J. (ed), 1986, Problems with the Report on Modula-2, Version 8, IST/5/13 Working Group paper N103, British Standards Institute.

7. Cornelius B.J., 1988, Problems with the Language Modula-2, Software—Practice and Experience, Vol 18, No. 6.

8. Fairley R., 1985, Software Engineering Concepts, McGraw Hill.

9. Goldberg A., Robson D., 1983, Smalltalk-80, The Language and Its Implementation, Addison-Wesley.

10. Gutknecht J., 1989, Variations on the Role of Module Interfaces, Structured Programming, Vol. 10, No. 1, Springer.

11. Henne E., et al., 1988, Modula-2 User Manual, p1 Gesellschaft für Informatik (German).

12. Keller R., 1989, Prototypingorientierte Systemspezifikation (Prototyping-Oriented System Specification), Verlag Dr. Kovac, Hamburg, (German).

13. Meyer B., 1989, From Structured Programming to Object-Oriented Design: the Road to Eiffel, Structured Programming, Vol. 10, No. 1.

14. Mössenböck H., Templ J., 1989, Object Oberon—A Modest Object-Oriented Programming Language, Structured Programming, Vol. 10, No. 4.

15. Moffat D.V., 1984, Some Concerns About Modula-2, Sigplan Notices, Vol. 19, No.12.

16. Pomberger G., 1986, Software Engineering and Modula-2, Prentice Hall.

17. Pomberger G., Bischofberger W., Keller R., Schmidt D., 1988, Topos - A Toolset for Prototyping-Oriented Software Development, Proceedings of the CGLA, Paris.

18. Pressman R.S., 1987, Software Engineering: A Practitioner's Approach, 2nd edition, McGraw-Hill.

19. Schmucker K., 1985, Object-Oriented Programming for the Macintosh, Hayden.

20. Sommerville I., 1985, Software Engineering, 2nd edition, Addison-Wesley.

21. Weinand A., et al., 1989, Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No. 2, Springer.

22. Wiener R., Sincovec R., 1984, Software Engineering with Modula-2 and Ada, John Wiley & Sons.

23. Wirth N., 1987, From Modula-2 to Oberon and the Programming Language Oberon, ETH Report, Zurich.

24. Wirth N., 1988, Programming in Modula-2, 4th edition, Springer.