THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY

A PARADIGM FOR TOP-DOWN DESIGN WITH PACKAGES

Vaclav Rajlich

CRL-TR-31-83

NOVEMBER 1983

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

A Paradigm for Top-down Design with Packages.

Vaclav Rajlich
Department of Computer and Communication Science
University of Michigan
Ann Arbor, MI 48109

Abstract.

The paper presents a paradigm for the design with packages. The paradigm combines the methodology of stepwise refinement with information hiding principle. It is based on steps of definition, decomposition, completion, and abstraction, which produce clean and well-defined packages. A part of the paradigm are the rules by which the procedures acquire the parameters (called first and second rules for parameters). The paradigm is illustrated by an example.

It is argued that "with" clauses of ADA (together with ADA order of compilation) encourage bottom-up programming while discouraging the top-down programming, and hence should be considered harmful.

1. Introduction.

Packages (sometimes also called modules) are generally recognized as an important feature of the new programming languages [1,6,13]. If properly used, they allow a program to be split into manageable pieces which can be individually designed, coded, and tested. They allow cooperation of several programmers on one software project. If they are powerfull and general enough, they extend the language by new constructs and hence they extend an applicability of the language. They are also intended to play a role of reusable parts for future programs.

These and other benefits were the reasons why packages or modules were included into ADA and other programming languages. The basic idea of the packages is information hiding [7]. Each package implements several objects like procedures, data structures, data types, etc. It is connected to the rest of the program through the so-called specification (in other languages also called interface). The specification lists all objects (procedures, functions, types, variables, etc.) which are defined in the body of the package and are available to the outside program. However the definitions themselves are invisible to the outside. Hence the implementation details and related complexities remain hidden inside the package.

During the process of program design, the role of the designer is to develop the specifications of the packages. All the beneficial reasons for using packages will materialize only when the packages are designed in a very careful manner. The experience shows that the difference between well-designed and poorly-designed packages may have an enormous influence on the size, effectiveness, and the cost of the resulting program.

It was argued that the design of the "right" packages is an extremely difficult art, particularly in the situation when the design is based mainly on designer's intuition. The designer has to forsee how his specifications are going to be implemented, which may be a formidable intelectual task. In fact, he is required to think far ahead, with all accompanying risks of omissions and unforseen circumstances.

It is our opinion that the succesfull paradigms have to offer very detailed design steps, and the steps should be independent of each other as much as possible. Also the paradigm should provide a rigorous methodology which will offer the firm guidance to the designer. When the designer mastered the detailed and rigorous process of the design, he can always return back to less rigorous and less detailed methodology, with the rigorous methodology in the background. Among the methods currently in the circulation, the designer has to choose either the principle of information hiding (as an example, see [2]), or

the rigor (as an example, see [5,11,14]), but not both. An overview of the currently used methodologies appears in [4].

The paradigm explained here is an extension of the stepwise refinement [12]. Stepwise refinement provides the most universal paradigm for program design, and it was chosen as the basis for the methodology of this paper. It was extended to include the principle of the information hiding, and hence to cover the design with packages. In particular, the "scope" is the basic unit of the design, rather than a procedure of [12]. Another important difference is the higher level of rigor as compared to [12]. In particular, we included the steps of completion and abstraction, and a thorough treatment of parameters (called "first and second reason for parameters").

The paradigm explained here has one important element of style: All variables are local in the resulting packages, i.e. they never appear as a part of a specification. The packages communicate via procedure calls and types only. This style is the most elegant style of modular decomposition of the systems, and it was widely advocated in the literature[13]. Also the resulting packages avoid nesting of procedures, which again was criticized in the literature [3,8].

The paper consists of four chapters. Chapter 2 contains the description of the methodology. It describes the basic steps of definition, decomposition, completion, and abstraction, on which the methodology is based. The two reasons for procedure parameters are introduced. Chapter 3 contains an example of the design in ADA-like design language. Chapter 4 contains some thoughts on conversion of the designs into programs. Chapter 5 contains a criticism of some ADA constructs from the point of view of the paradigm.

2. Basic notions.

During the program design, the program consists of two parts: existing part, which is the part already finished (and possibly stored in the computer), and intended part, which consists of everything what has not been done yet [10]. The role of a step of the design methodology is to add to the existing part, and to subtract from the intended part.

There is an interface between the two parts of the program, which will be called backlog interface. It consists of the objects which have already been used in the existing part, but which were not defined. These objects will be called unfinished objects. They are:

(i) Procedures/functions which have been called but whose bodies have not been defined

(ii) Types which already have been used but have not been

defined

(iii) Variables whose names have been introduced but their types have not been declared.

The unfinished objects are interrelated by the following relations:

(i) Relations "read", "write", and "readwrite" between variables and procedures/functions

(ii) Relations "in", "out", and "in out" between unfinished types and procedures/fuctions. This relation describes the mode and type of formal parameters of procedures/functions.

Examples of backlog interfaces are in Fig. 2 and Fig. 3. In the figures, the procedures/functions are denoted by rectangles, the variables are denoted by ovals, and the types are denoted by diamonds. Relations are denoted by arrows pointing in the direction of the information flow. Procedures, functions, and types also contain the names of all packages in which they were used.

Scope of variable or type V is the set of all procedures/functions related to V. Scope of a procedure P is the procedure itself.

The methodology of this paper is based on the following steps:

(i) definition
(ii) decomposition and completion
(iii) abstraction.

Each of these steps removes certain objects from the backlog interface and may add new ones to it. Each of them also adds one package to the existing part of the program. The steps are described in the following way:

(i) Definition.

Definition is a step in which we define one or several unfinished objects in terms of the primitives of the programming language. If the object is a procedure, we will define its body. If it is a variable, we will give its type. If it is a type, we will define its representation. These objects are then removed from the backlog interface.

An important property of definition is that the smallest unit of definition is a scope. Every definition step means that one or several scopes are defined at once.

A procedure can be fully defined only if all related objects (variables, types) are defined in the same step. If procedure P is related to variables V1, V2, ..., Vn, and to types T1, T2, same step, then procedure P can be fully defined. If one or more variables and/or types remain undefined, then procedure P can be defined only partially. It's body will have to contain calls of new unfinished procedures P1, P2, ..., Pk which will be related to the variables/types which remained undefined.

An important rule is the rule which we will call <u>first reason for parameters</u>. It is explained in the following way: Let P be a procedure which is related to variables V1, V2, ..., Vn where variables V1, V2, ..., Vm were defined, variables V(m+1), defined with new unfinished procedure P1 being called in its body. The procedure P1 will have actual parameters V1, V2, ..., Vm, i.e. the body of P will be

```
procedure P1 is
      begin
            .
            .
            .
            P1(V1, V2, ..., Vm)
            .
            .
            .
      end P1;
```

The rule can be justified by the following reasoning: Each definition step produces a package. As remarked earlier, we allow the communication between packages through procedure parameters only, disallowing direct accessibility of data. Procedure P1 may inherit all relationships of the original procedure P, i.e. it may be related to all variables V1, V2, ...,Vn. However the definitions of these variables will be spread over at least two packages: The current package where V1, V2, ...,Vm are defined, and some future packages where both P1 and some of the variables V(m+1), ..., Vn will be defined. Then there is the need for the parameters P1(V1, V2, ...,Vm). As an example of a step where first reason for parameters was applied, see package DEFINE_D of Sectin 3.

(ii) <u>Decomposition and completion.</u>

Decomposition and completion is another step of the methodology. In it, we decompose an unfinished object A into several smaller unfinished objects A1, A2, ..., An. If object A is a variable, type, and procedure, then objects A1, A2, ..., An will also be variables, types, and procedures, respectively. Moreover if A was related to B, then a nonempty subset of A1, A2,

The step of decomposition is usually started with a decomposition of a variable or a type. Then the related procedures/functions are decomposed. Then the completion follows.

In the substep of completion, we inspect all new unfinished objects and try to determine whether they can function correctly or whether they need to be related to some additional objects. There are two situations which require introduction of new objects: First, two procedures may need to communicate with each other, and hence there is a need for a new variable which will facilitate this communication. Second, variables may need an initializing procedures which will set the initial values. None of these new objects were introduced by decomposition, hence we have to have the special substep of completion. An existing part of the program is <u>complete</u> when no new objects can be introduced by the process of completion, i.e. all communications among

procedures have been served by appropriate variables, and all variables have been properly initialized.

The step of decomposition and completion again produces a package.

(iii) Abstraction.

Abstraction is a special case of definition in which a variable V is declared to be of type T, where T is an unfinished type. As in every definition, the whole scope of variable V has to be defined. If V is read by procedure P, then the body of P will call a new unfinished procedure P1(V:in T). This action will be called second reason for parameters.

The purpose of abstraction step is to replace several similar unfinished variables by one type, and similar unfinished procedures by one more abstract procedure. The reason is economical: backlog interface will be simplified, because several variables and procedures will be replaced by a smaller number of newly introduced types and procedures, respectively.

As shown in the example of the next section, the procedures or functions may acquire several parameters either through first or second reason or both. Hence if we have procedure P(V1,V2,...Vn) which aready has several parameters and it is in scope of a variable W which is being defined, then its body will contain a call of procedure P1(V1,V2,...,Vn,W).

When packages are produced by these steps, then the resulting program consists of packages organized by the relationship of procedure calls into a directed acyclic graph. In [9], this relationship was called the relationship of "seniority". The resulting program consist of several layers of virtual machines, where the formerly defined backlog interfaces have become the virtual machines.

The design methodology desribed by these steps is directly applicable to all module or package-oriented programming languages, including new modular languages like ADA, Modula-2, or MESA [1,6,13]. In the next section, the methodology will be explained on an example, using an ADA-like language for the demonstration.

3. An example.

In this section, we shall illustrate the paradigm on the following geometrical problem:

Find the intersection of two lines LN1 and LN2, and find two points A, B on line LN1 with distance 1.0 from the intersection, see Fig. 1.

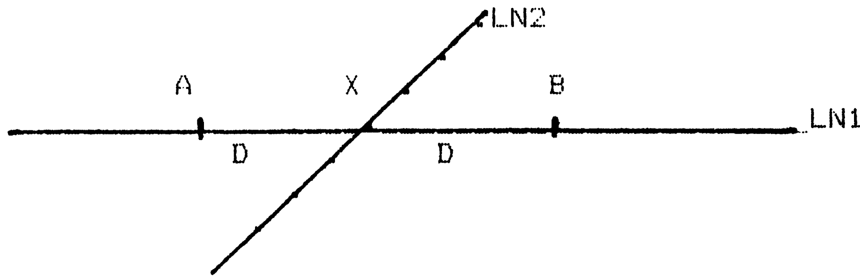

Fig. 1

The problem will be solved in this section with the use of an ADA-like design language. It should be immediatelly understandable to anyone familiar with ADA or any other package or module-oriented higher-level language. The "used in" clause precedes every package definition. It consists of an executable part which is almost identical to ADA, and a graphical part which describes backlog interface. The main difference with ADA is the occurance of "used in" clause preceding each package. The clause lists all packages which use the objects of the given package.

As the start, the whole problem is solved as a call of one procedure:

```
procedure MAIN is
     begin
          MAIN1;
     end;
```

The procedure MAIN1 is decomposed in the following way:

```
used in MAIN;
package DECOMPOSE_MAIN1 is
     procedure MAIN1;
end DECOMPOSE_MAIN1;
package body DECOMPOSE_MAIN1 is
     procedure MAIN1 is
          begin
               READ_LN1;
               READ_LN2
               READ_D;
               INTERSECTION;
               FIND_PTS;
               WRITE_A;
               WRITE_B;
          end MAIN1;
     end DECOMPOSE_MAIN1;
```

In this text, READ_LN1, READ_LN2, and READ_D read input data for lines LN1, LN2, and distande D, respectively. Procedure INTERSECTION finds the intersection of LN1 and LN2. Procedure FIND_PTS finds points A and B, and procedures WRITE_A and WRITE_B print them out. "Used in" clause indicates that procedure MAIN1 is called in subprogram MAIN.

In the subsequent completion, we will introduce variables LN1, LN2, D, X, A, and B, which will facilitate the communication between the procedures. Their role is the same as in Figure 2. They will not appear as a part of the executable program, but will become a part of backlog interface of Fig.2.

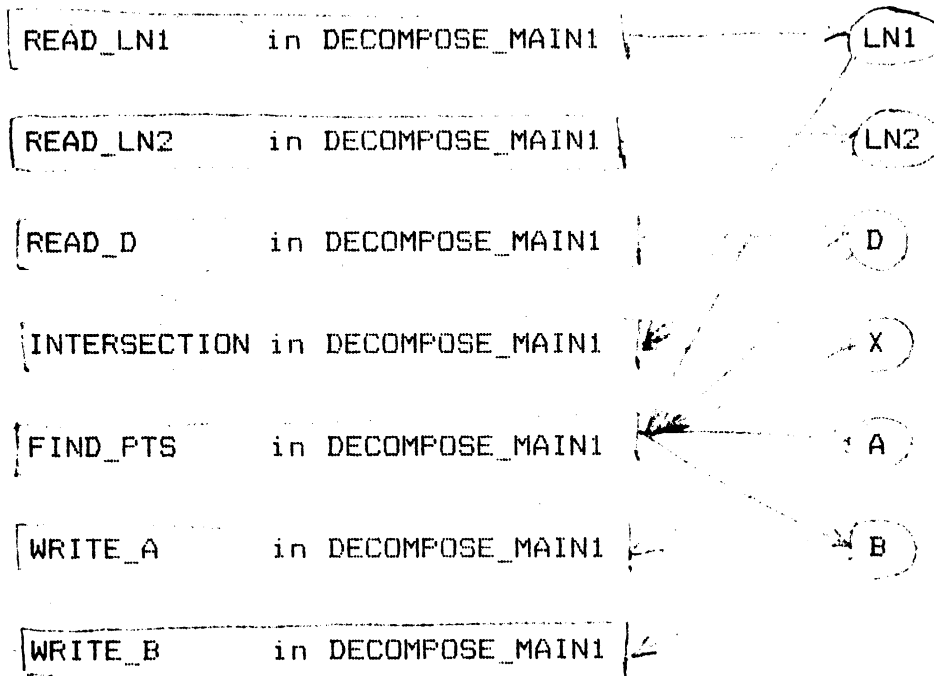

Fig. 2.

Note that backlog interface contains some additional information which does not appear in the previous package, like names of the variables, their relationship to procedures, etc.

Next step is definition of D, which will produce the following package:

```
used in DECOMPOSE_MAIN1;
package DEFINE_D is
      procedure READ_D;
      procedure FIND_PTS;
end DEFINE_D;
package body DEFINE_D;
      D:float;
      procedure READ_D is
            begin
                  put("Enter the distance:");
                  get(D);
            end READ_D;
      procedure FIND_PTS is
            begin
                  FIND_PTS(D);
            end FIND_PTS;
end DEFINE_D;
```

In this step, we defined variable D and procedures READ_D, FIND_PTS of its scope. In case of procedure READ_D, we were able to define the body of the procedure completely using the standard

constructs of the underlying language ADA. In case of procedure FIND_PTS, observe that the procedure is in scope of several variables: D, LN1, A, B. Only one of these variables is being defined in this step, hence the body of the procedure must contain a call to a subprocedure with actual parameter D, following the first reason for parameters. By a coincidence, the call of the subprocedure is the only statement of the body of FIND_PTS. The subprocedure was named FIND_PTS(D:in float), i.e. it reuses the name of the original procedure, which is allowed by the principle of overloading in ADA. "Used in" clause indicates where the procedures of the package are called. Note that the information in backlog interface of Fig. 2 was useful for both determining the scope of D and determining the "used in" clause. New backlog interface is in Fig. 3.

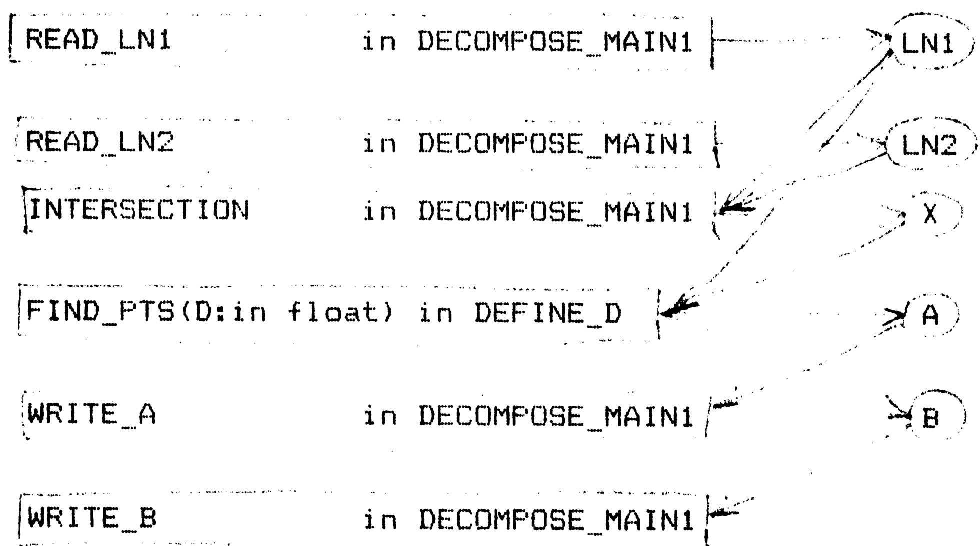

Fig. 3.

Next step is a step of abstraction. We observe that variables LN1 and LN2 are of the same type, and hence can be defined in the following package:

```
used in DECOMPOSE_MAIN1;
package ABSTRACT_LN is
     procedure READ_LN1;
     procedure READ_LN2;
     procedure INTERSECTION;
     procedure FIND_PTS(D:in float);
end ABSTRACT_LN;
package body ABSTRACT_LN is
     LN1,LN2:LN;
     procedure READ_LN1 is
          begin
               READ_LN(LN1);
          end READ_LN1;
     procedure READ_LN2 is
          begin
               READ_LN(LN2);
          end READ_LN2;
     procedure INTERSECTION;
          begin
               INTERSECTION(LN1,LN2);
          end INTERSECTION;
     procedure FIND_PTS(D;in float) is
          begin
               FIND_PTS(D,LN1);
end ABSTRACT_LN;
```

Note that in this step, the second reason for parameters was applied in definition of all procedures of the package body. New undefined type LN appears in the current backlog interface of Fig.4.

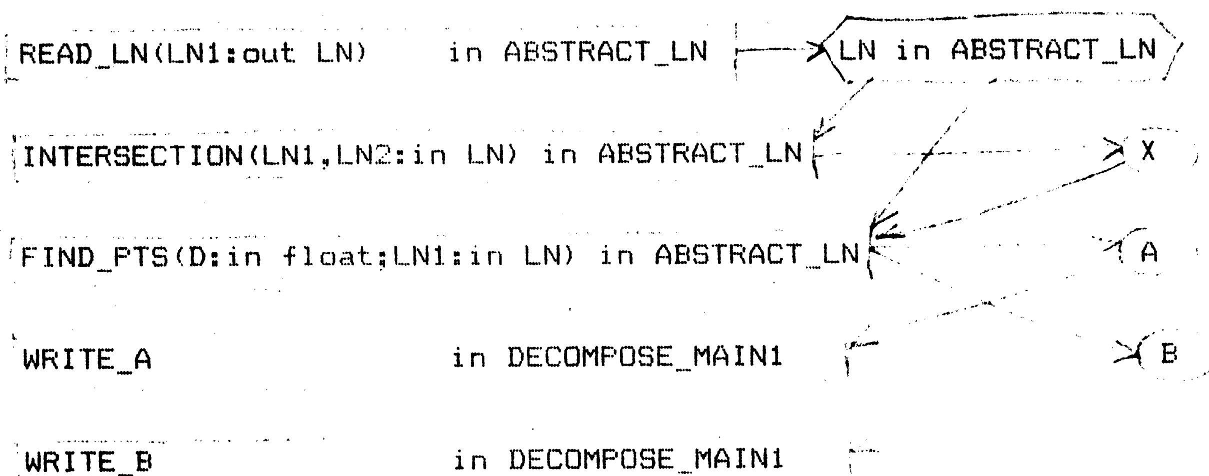

Fig. 4.

Next step is again a step of abstraction, in which points A, B, and X are declared to be of the same type. In the step, all procedures of scopes of A, B, and X are also defined:

```
used in DECOMPOSE_MAIN1, ABSTRACT_LN;
package ABSTRACT_PT is
      procedure INTERSECTION(LN1,LN2:in LN);
      procedure FIND_PTS(D:in float;LN1:in LN);
      procedure WRITE_A;
      procedure WRITE_B;
end ABSTRACT_PT;
package body ABSTRACT_PT is
      X,A,B:PT;
      procedure INTERSECTION(LN1,LN2:in LN) is
            begin
                  INTERSECTION(LN1,LN2,X);
            end INTERSECTION;
      procedure FIND_PTS(D:in float;LN1:in LN;A,B:out PT);
            begin
                  FIND_PTS(D,LN1,X,A,B);
            end FIND_PTS;
      procedure WRITE_A;
            begin
                  WRITE_PT(A);
            end WRITE_A;
      procedure WRITE_B;
            begin
                  WRITE_PT(B);
            end WRITE_B;
end ABSTRACT_PT;
```

The new backlog interface is in Fig. 5.

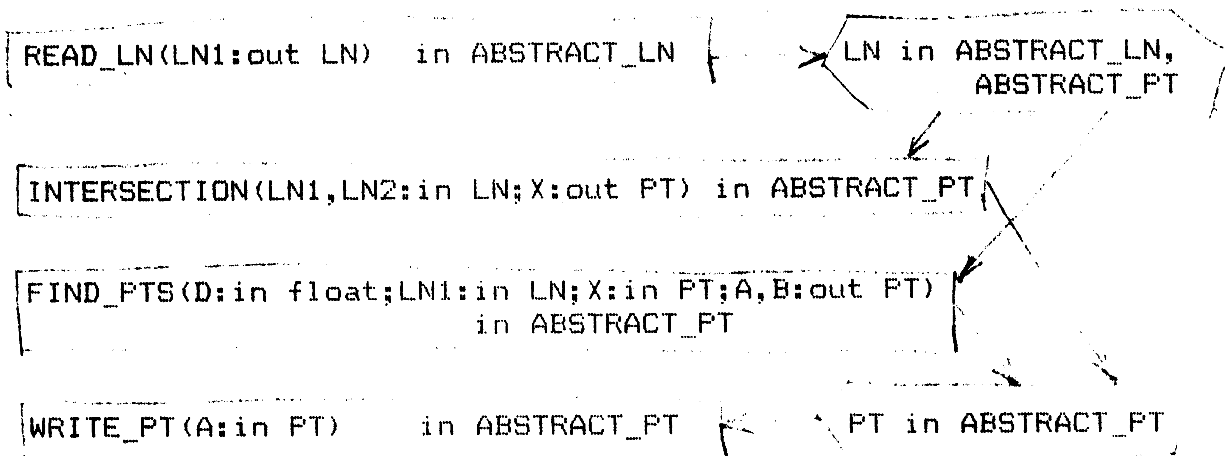

Fig. 5.

In the last step, we will define both PT and LN. An alternative option would be to define only one of them and postpone the definition of the other to the following step. We will give only specifications here; The body of the package is left to the reader.

```
used in ABSTRACT_LN, ABSTRACT_PT;
package DEFINE_ALL is
     type LN is private;
     type PT is private;
     procedure READ_LN(LN1:out LN);
     procedure INTERSECTION(LN1,LN2:in LN;X:out PT);
     procedure FIND_PTS(D:in float;LN1:in LN;X:in PT;
                              A,B:out PT);
     procedure WRITE_PT(A:in PT);
private;
     type LN is record
                    X_SECTION,Y_SECTION:float;
                end record;
     type PT is record
                    X,Y:float;
                end record;
end DEFINE_ALL;
```

4. Conversion of designs into programs.

The paradigm explained in the previous two sections offers a rigorous method of program design. It is obvious that the designs are already very close to the future programs. In fact, there are only two deficiencies which have to be removed:

(i) The packages are very small.

(ii) The packages contain "used in" clauses which do not have any counterpart in ADA.

The first deficiency is removed by merging several steps of the methodology into one so that a reasonably sized packages are achieved. This is done by a textual processing of the design of in the following way: Suppose that we have two design packages A and B which we want to merge together, then we will macroexpand bodies of procedures and definitions of types of B in places of procedure calls and variable declarations in A, respectively. Moreover we will copy variable declarations and "used in" clauses from both A and B into the resulting package.

To remove the second deficiency, note that "used in" relationship is an inverse relationship to the combination of "with" and "use" clause of ADA. Hence if we have two packages P and R which are related in the design in the following way:

```
used in ...;
package P;
      .
      .
      .
end P;

used in P,...;
package R;
      .
      .
      .
end R;
```

then replacing "used in" by a combination of "with" and "use" will give us

```
with R,...; use R, ...;
package P;
      .
      .
      .
end P;

with ...; use ...;
package R;
      .
      .
      .
end R;
```

Let us demonstrate the process of conversion on the design from the previous section. As the first step, we may merge procedure MAIN with packages DECOMPOSE_MAIN1 and DEFINE_D to produce one compilation unit called NEW_MAIN. Similarly packages ABSTRACT_LN and ABSTRACT_PT may be merged into one package ABSTRACT. Then we will replace "used in" clauses by "with" and "use" clauses. The resulting program in ADA is the following one:

```
with ABSTRACT; use ABSTRACT;
procedure NEW_MAIN is
      D:float;
      begin
            READ_LN1;
            READ_LN2;
            get(D);
            INTERSECTION;
            FIND_PTS(D);
            WRITE_A;
            WRITE_B;
      end NEW_MAIN;
```

```
with DEFINE_ALL; use DEFINE_ALL;
package ABSTRACT is
      procedure READ_LN1;
      procedure READ_LN2;
      procedure INTERSECTION;
      procedure FIND_PTS(D:in float);
      procedure WRITE_A;
      procedure WRITE_B;
end ABSTRACT;
package body ABSTRACT is
      LN1,LN2:LN;
      X,A,B:PT;
      procedure READ_LN1 is
            begin
                  READ_LN(LN1);
            end READ_LN1;
      procedure READ_LN2 is
            begin
                  READ_LN(LN2);
            end READ_LN2;
      procedure INTERSECTION is
            begin
                  INTERSECTION(LN1,LN2,X)
            end INTERSECTION;
      procedure FIND_PTS(D:in float) is
            begin
                  FIND_PTS(D,LN1,A,B);
            end FIND_PTS;
      procedure WRITE_A is
            begin
                  WRITE_PT(A);
            end WRITE_A;
      procedure WRITE_B is
            begin
                  WRITE_PT(B);
            end WRITE_B;
end ABSTRACT;

package DEFINE_ALL;
      --Both specification and body of DEFINE_ALL
      --is unchanged; however we may need to specify I/O
      --environment by an additional "with" clause.
```

5. Use and with clauses of ADA considered harmful.

Relationship of compilation units in ADA is expressed by "use" and "with" clauses. They specify what other compilation units are necessary for the execution of a given unit in which the clause appeared, and they also determine the order of compilation.

However the previous sections demonstrated that the "with" and "use" clauses of ADA create a special problem when designing

program by top-down method. The reason is that during the top-down design, we may know what procedures, variables, or types are needed, but we do not know which compilation unit they will be defined in. Hence the contents of the "with" and "use" clauses can be determined only after the whole program (or most of the program) was designed.

This contrasts sharply with the bottom-up design. The paradigm of bottom-up design is based on the idea of extension of the existing language by additional constructs. For this purpose, "use" and "with" clauses are completely natural, because lower compilation units are designed before the higher units are. Then there is no problem to determine in which compilation unit the procedures, types, etc. are defined, i.e. it is possible to determine "with" and "use" clauses immediatelly.

Hence we have to conclude that "with" and "use" clauses (and the order of compilation in Ada) encourage bottom-up programming, while discouraging the top-down programming in a substantial way. It was argued many times that the top-down paradigm is prefferable to the bottom-up one [12]. Hence these two programming constructs encourage an inferior programming practice, and should be considered harmful.

References.

[1] Ada Programming Language, Military standard MIL-STD-1815.

[2] Booch, G., Software Engineering with Ada, The Benjamin Cummings Publ. Co, Menlo Park, CA, 1983.

[3] Clarke, L.A., Wileden, J.C., Wolf, A.L., Nesting in Ada Programs is for the Birds, Sigplan Notices Nov. 1980, 139-145.

[4] Freeman, F., Wasserman, A.I., Software Development Methodologies and Ada, Res. Rep., The University of California, Irvine, CA.

[5] Jackson, M., System Development, Prentice-Hall, Englewood Cliffs, NJ., 1983.

[6] Lauer, H.C., Satterthwaite, E.H., The Impact of Mesa on System Design, Proc. 4th. International Conference on Software Engineering, IEEE Catalog No. 79CH1479-5C, 1979, 174-182.

[7] Parnas, D.L., Designing Software for Ease of Extension and Contraction, IEEE Trans. on Software Engineering, March 1979, 128-137.

[8] Rajlich, V., Hierarchical vs. Block Structure, Information Processing Machines, Vol. 21, Academia Prague 1979, 23-33.

[9] Rajlich, V., Problems of Module Interconnection Language, in Hibbard, P.G., Schuman, S.A., Constructing Quality Software, North-Holland 1978, 147-152.

[10] Rajlich, V., Stepwise Refinement Revisited, Res. Rep. CRL-TR-13-83, Computing Research Laboratory, University of Michigan, Ann Arbor, MI.

[11] Ross, D.T., Schoman, K.E.Jr., Structured Analysis for Requirements Definition, IEEE Trans. on Software Engineering, Jan. 1977, 6-15.

[12] Wirth, N., Program Development by Stepwise Refinement, Communications of ACM, April 1971, 221-227.

[13] Wirth, N., Modula-2, Res. Rep. 36, ETH, Institut fur Informatik, Zurich, March 1982.

[14] Yourdon, E., Constantine, L.L., Structured Design, Prentice-Hall, Englewood Cliffs, NJ., 1979.