THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[*]

PARADIGMS FOR DESIGN AND IMPLEMENTATION IN ADA[**]

Vaclav Rajlich

CRL-TR-43-84

October 1984

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

[*]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

[**]Ada is a registered trademark of the United States Government, Ada Joint Program Office (AJPO)

Abstract.

In the paper, three characteristic paradigms of a design and implementation in Ada are described and compared: Bottom-up incremental programming, top-down semiincremental programming, and large-small traditional programming. Several examples illustrate the selection of the correct paradigm for the given set of circumstances.

1. Introduction.

Programming languages Ada [1] and others [2], [8], [15], [18], represent a new generation of languages. They are a culmination of long research in modularity, i.e. the tools and techniques of decomposition of large software systems into smaller pieces. They are also new and powerful software tools, providing extensive help to both the designer and programmer who uses them.

One of the most powerful modularity techniques is the principle of information hiding as outlined by Parnas and others [10]. The principle can be briefly summarized in the following way: We want to decompose the programs into modules (packages) which will have simpler and more understandable interfaces than the detailed code they are hiding inside. They will be reusable, will localize the impact of the changes in the program, allow cooperation of several programmers on a program, etc.

Traditionally, the process of program creation is divided into two phases, the design and implementation, where the purpose of design is to find the "right" packages with the properties described above, while purpose of the implementation is to actually produce the code for the packages. The programming language Ada contains elaborate features which support the implementation (i.e. code writing) of programs divided into packages. In this paper, Ada will be used both as a design and implementation language, and several paradigms will be discussed.

In Section 2, we describe separate compilation in Ada. Section 3 describes a certain desirable architecture of Ada programs. Section 4 describes paradigms and life-cycles in general, with the emphasis on non-traditional incremental and semi-incremental life-cycles. In Sections 5, 6, and 7, we describe three selected paradigms. The summary in Section 8 contains several hypothetical examples of a paradigm selection.

2. Separate compilation in Ada.

The basic building block of Ada programs is a compilation unit; the whole program consists of one or more compilation units [1]. We shall avoid discussing the complexities of compilation units in general, and shall concentrate on packages instead; however the paradigms discussed in this paper can be applied to subprograms and generics, too.

Each package is associated with two compilation units: package specification and package body. A package specification is a list of all declarative items (i.e. objects, types, subprograms, tasks, exceptions, etc.) which are implemented in the package and can be used by other packages of the program. A package body contains all relevant definitions. Moreover, a package body can contain other entities, invisible from the outside. Hence the package specification is a "window" of a limited size, which allows only a limited number of the entities of the package to be seen, hiding all the remaining ones. Moreover it hides the implementation details of the visible objects. This principle is called "information hiding principle" in the literature [10], and it allows decomposition of large systems into smaller packages, where each package and its interconnections are simpler than the whole program. Hence a programmer who deals with one package does not have to know the whole program, and his task is substantially simplified.

A typical program consists of several packages. It must contain both specification and body of each package. Packages can be interconnected by either with or separate clauses, which determine both the relationships and the order of the compilation.

Suppose we have two packages X and Y interconnected by the following "with" clause:

```
with X; package Y is ...,
```

Then the order of compilations is given by the following rules:

(i) package specifications of both X and Y must be compiled before package bodies are compiled,

(ii) specification of X must be compiled before specification of Y is compiled.

Hence the possible orders of compilation of packages X and Y are:

specification X, body X, specification Y, body Y
specification X, specification Y, body X, body Y
specification X, specification Y, body Y, body X.

Besides the order of compilation, the "with" clauses determine which package can use items of which package. In the example above, package Y is allowed to use items listed in the specification of X.

The "with" clauses organize the packages of a program into a directed acyclic graph (dag).

Another way how to organize packages is to use the "separate" clauses. Suppose we have the following program:

```
package Y is ... end Y;
package body Y is
     package D is ... end D;
     package body D is separate;

...

end Y;
```

```
separate (Y);
package body D is ... end D;
```

There is only one order of compilation: specification Y, body Y (includes specification D), body D. Visibility rules are given by the nesting of D in Y. The "separate" clauses interconnect the packages into a tree-structured hierarchy.

One program can have the packages interrelated by both "with" and "separate" hierarchies.

3. Hierarchical architecture for Ada programs.

In the literature, two orthogonal hierarchical architectures for computer programs were described. One deals with the organization of programs into "layers", where interfaces among layers are the so-called "virtual machines" [5], and it will be called the seniority hierarchy. The other hierarchy deals with the decomposition of larger objects into smaller parts, and will be called the parent-child hierarchy. A common misconception is to equate these two hierarchies. We shall follow the style of [15] and make a distinction between the two hierarchies.

The seniority hierarchy provides an intuitive basis for two important paradigms: top-down and bottom-up. This hierarchy requires an organization of the packages into directed acyclic graphs [12], and hence it is natural to define it by the "with" clauses of Ada. The virtual machines are then the "cuts" of the dag. Hence if we have the situation

```
with X; package Y is ...;
```

then Y will be called senior (to X), and X will be called junior (to Y).

It should be noted that the "separate" clauses were suggested by some researchers to serve as a tool for the seniority hierarchy [16], particularly in the situation when top-down programming is used. However they are not suitable for this purpose; they force tree structure on the system, which constrains the architecture in a substantial way. Example in [14] illustrates the situation where the tree structure is inadequate. We will use "separate" clauses for a different purpose.

The parent-child hierarchy is intuitively characterized by big packages divided into smaller ones. This hierarchy requires a tree-like structure. It is here where the "separate" clauses can be utilized. If we have a situation

```
separate (B); package body A is ...;
```

then B will be called a parent (of A), and A will be called a child (of B).

The two hierarchies are orthogonal to each other and can be combined in one program. However in that case, a certain element of style should be observed: The children of a common parent should also be ordered by the relation of seniority, even if Ada does not require such ordering. Then the children will also create a hierarchical decomposition into layers and virtual machines, which will make it possible to design and program them both top-down and bottom-up.

The two hierarchies provide a natural selection of relationships among the packages. Moreover they provide four basic directions, in which paradigms for the design and implementation may progress: Bottom-up, which progresses in the direction from the junior to the senior packages, top-down which progresses in the exactly opposite direction, i.e. from the senior to the junior packages, large-small paradigm which follows the direction of the "separate" hierarchy, where parent packages are written before the child packages, and the "small-large" paradigm, where child packages are written before the parent packages.

It should be immediatelly noted that the small-large paradigm is incompatible with both modern programming practices and Ada. It was widely used in the past with less sophisticated programming languages, when separate modules were written by different programmers, and then later "integrated" into one program. It was typical to discover substantial discrepancies and misunderstandings among the programmers during the integration phase, which required substantial revisions in the code. Hence integration was the least predictable and potentially costly phase of the software life-cycle. Because of this cost and uncertainty, and because all other paradigms are superior to small-large paradigm in all respects, it should no longer be used, and is not discussed further in this paper.

4. Life-cycles.

The paradigms of the previous section should be distinguished from the orthogonal notion of software "life-cycle". The life-cycle is a sequence of the stages through which a piece of software has to pass during its life. In this paper, we shall talk about three different lifecycles: Traditional, incremental, and semi-incremental.

The *traditional* software life-cycle is well documented in the literature [7], [9], [11], and consists of the following phases:

Requirements specification, where the task is described in a language understandable to both the implementors and the users (answering the question *what* is to be implemented). An interesting part of the requirements specification is the verification of the specifications via rapid prototyping [20]. As interesting and important the as specifications are, we shall not deal with them in this paper and shall rather concentrate on the remaining portions of the life-cycle.

Design, where the algorithms are defined and the system is decomposed into packages (answering the question how is the system going to be implemented).

Implementation, where the actual code is written, integrated, and tested.

Evolution, where the system is being operated and modified according to changing environment or whenever errors are discovered.

In the traditional life-cycle, the complete design of the program or the system must be finished, before the implementation starts. Typically, the design language will be completely different from the implementation language. The methods covering the traditional paradigm are well known [11]. In correspondence to [21], the result of the design phase will be called a workproduct.

However Ada allows a departure from this sequence, because it can be used both as a design and implementation language. The departures we will talk about here will be called incremental and semi-incremental life-cycles.

A characteristic feature of the incremental life-cycle is that it uses one language for both design and implementation, and hence it merges the steps of design and implementation into one (for simplicity, we shall use word "implementation" to denote this merged step). During the implementation, the program typically is incomplete, i.e. it consists of two parts: the existing part, which is the actual program so far stored in the computer, and the intended part, which is everything not yet written. At the beginning, the existing part is empty and the whole program is intended. At the end, the intended part is empty and the whole program is existing. The program implementation is a sequence of incremental steps, each adding something to the existing part and deleting something from the intended part.

An advantage of the incremental life-cycle is the possibility of verification of incomplete designs. Lack of this possibility is one of the major problems with the traditional life-cycle. Another advantage is the saving in the volume of documentation the designers/programmers must produce. A careful scrutiny of most design documents reveal that with the traditional paradigm, the design and implementation overlap to a substantial degree.

The incremental life-cycle requires support by software tools. The programming language must contain constructs which will allow the incremental steps, where the code written for one step will remain valid for all the remaining steps (unless the programmer changes his mind). It also requires a verification method for incomplete programs.

The semi-incremental life-cycle is half-way between incremental and traditional life-cycles. It requires separate phases of design and implementation, but these phases can be substantially overlapped. This means that the design language can be easily translated into the programming language, and hence the design at any stage can be easily converted into code and tested. This code may require some modifications when additional steps of design are added, but the modifications are minor and mechanical. Also the semi-incremental life-cycle may require a special work-product, which is a document prepared for the purpose of the design only, not a part of the code. Also there must be methods of verification of incomplete designs translated into code.

Ada is a software tool which supports several combinations of paradigms and life-cycles. In the following sections, we shall explore three most characteristic combinations: bottom-up and incremental, top-down and semi-incremental, and large-small and traditional. There are other possibilities like traditional life-cycle with top-down design and bottom-up implementation; brief comment related to these possibilities is in Section 8.

5. Bottom-up incremental programming.

The bottom-up paradigm is based on the intuitive idea of "language extentions", and it is a well-known paradigm. The basic idea is described in the following way: Suppose we want to solve a problem, which can be easily solved using a certain set of operations, objects, types, etc. These entities however are not available in Ada. Then we shall define a package which will provide them, and use this package as the first step towards the solution of the problem. Intuitively, we see the packages as steps in the extension of Ada, which provide increasingly complex entities needed for the solution. This process of extention may require several steps, where we extend Ada with entities of one package, and then use these entities in definition of another package which has even better entities, etc., until the problem is solved.

In the Appendix, there is an example of a solution to the eight queens problem. In the bottom-up approach, package COLUMN is defined first. The package provides type COLUMN and all necessary procedures which deal with it. Then the package BOARD is defined, and finally the problem is solved by definition of MAIN.

The programming language Ada provides all necessary constructs and the order of compilation to support the incremental life-cycle in combination with the bottom-up paradigm. There is no need to change the previously defined packages when a new package is added.

The advantages of this programming combination are the following:

- There is no extra design document necessary, hence no extra cost associated with it. In fact, the design is done simultaneously with the implementation, directly in Ada.

- The paradigm is very forgiving. If a wrong decision is made at any turn, it does not invalidate the design. It may mean that the the design will be less optimal, containing more code than necessary, but it usually does not mean that we have to redo the whole design from scratch.

However the programming combination also has disadvantages:

- A well-known disadvantage of the bottom-up approaches is that there is no guarantee that the final product will satisfy the specifications, or that it will be well balanced. Some authors considered this disadvantage to be so severe that they ruled out the bottom-up paradigm as generally unsuitable [17].

- The paradigm allows only a limited number of people to work on a project.

- The paradigm is rather difficult. It is a highly creative act to discover items (i.e. objects, types, etc.) which will be useful for the solution of a particular problem.

6. Top-down semi-incremental programming.

The particular methodology on which the top-down paradigm is demonstrated is the refinement methodology (RM) of [13] and [14]. Other top-down methodologies can be found in [4], [6], [17], [19], [21], etc., but none of them leads to a semi-incremental life-cycle.

The top-down paradigm starts with specifications, which describe the problem to be solved, and serve as a departure point for the design. An important workproduct of RM is the so-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. Typically, 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 interconnected into data-flow graphs 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.

As in [14], we shall represent backlog interfaces by graphs, where 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 are used. An example of backlog interface is in Fig. 1.

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

The RM methodology 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 tentative 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, T3..., Tm and all these variables and types were defined in the 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. In this situation, the so-called _first reason for parameters_ has to be applied. 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), ..., Vn were left undefined (m<n). Then P will be partialy 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 P is
          begin
               .
               .
               .
               P1(V1, V2, ..., Vm);
               .
               .
               .
          end P;
```

This rule can be justified by the following reasoning: Each definition step produces a package. For reasons of style, 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, V3,...,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).

(ii) Decomposition and completion.

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, A3,..., An will be related to B by the same kind of relations.

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 initializing procedures which will set the initial values. None of these new objects were introduced by decomposition, hence we must have the special substep of completion. An existing part of the program is complete 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 is called second reason for parameters.

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

The packages produced by the steps of RM can be directly written in Ada, except for one thing: The "with" clauses cannot be utilized, because it is unclear at the time of introduction, in which package the individual items of the backlog interface will be located. Hence we use the "used in" clauses, which specify where the items are used instead. The "used in" clauses are the only difference between the actual Ada code and the design. For more details on RM, see [14].

To convert designs into code, note that the "used in" relationship is an inverse relationship to the combination of the "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:

```
package P is ... end P;

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

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

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

package R is ... end R;
```

Let us illustrate the programming on the example in the Appendix. In that example, the first compilation unit to be implemented is procedure MAIN. The procedure calls several procedures, which in turn communicate with each other through a data structure BOARD. Hence after the first step, the backlog interface is defined by the graph of Fig. 1.

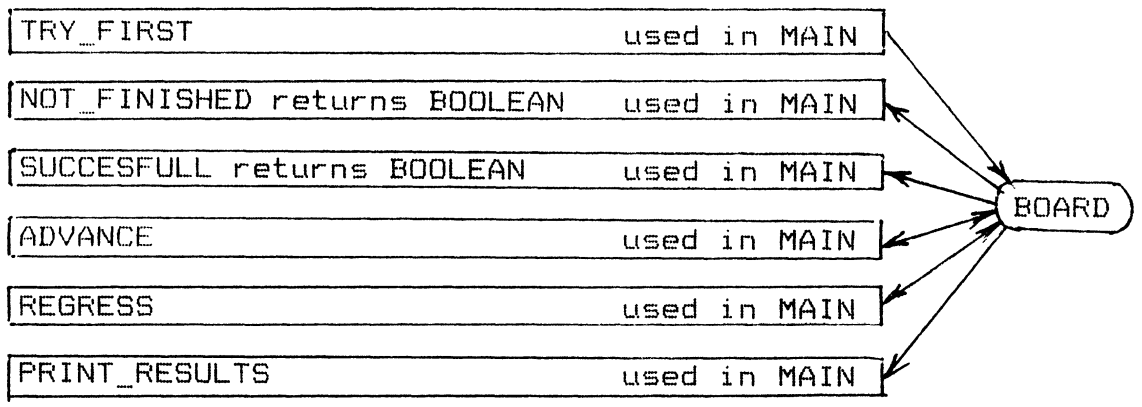
TRY_FIRST used in MAIN
NOT_FINISHED returns BOOLEAN used in MAIN
SUCCESFULL returns BOOLEAN used in MAIN
ADVANCE used in MAIN
REGRESS used in MAIN
PRINT_RESULTS used in MAIN
BOARD

Fig. 1

In the next step, we shall deal with the scope of the object BOARD, and define package BOARD. The object BOARD will become a local object of the package, defined as an array(1..8) of COLUMN, where COLUMN is an unknown type. Both package specification and body will be defined simultaneously:

```
used in MAIN;
package BOARD is ... end BOARD.
```

The backlog interface appears in Fig. 2.

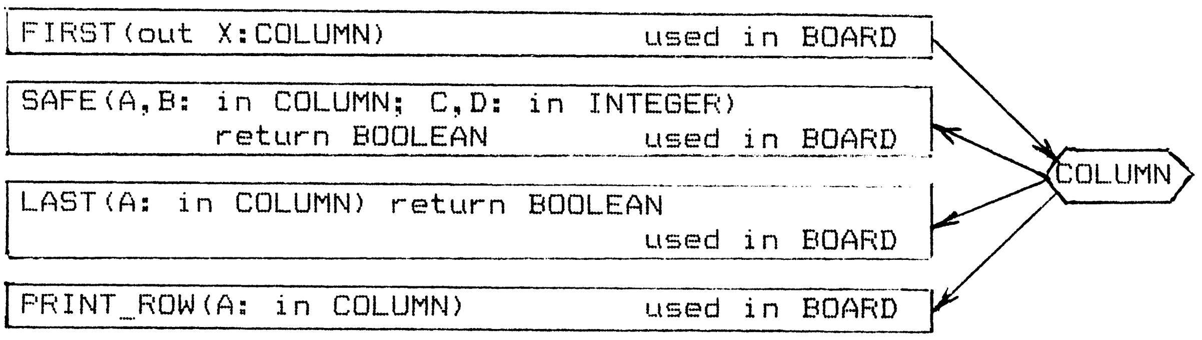
FIRST(out X:COLUMN) used in BOARD
SAFE(A,B: in COLUMN; C,D: in INTEGER) return BOOLEAN used in BOARD
LAST(A: in COLUMN) return BOOLEAN used in BOARD
PRINT_ROW(A: in COLUMN) used in BOARD
COLUMN

Fig. 2.

In the last step, we shall define the scope of the type COLUMN. This is done in package COLUMN of the Appendix. The package (both the specification and the body) will have the form:

```
used in BOARD;
package COLUMN is ... end COLUMN;
```

After this step, the backlog interface is empty, and hence we are finished with the design. In order to convert the design into code, we have to convert the "used in" clauses into a combination of "with" and "use" clauses. The resulting program appears in the Appendix. The conversion is straighforward and mechanical, and can be done after each package is added. Hence it passes our criteria for the semi-incremental life-cycle.

The advantages of the top-down semi-incremental programming combination are:

- Since the specifications are the departure point for the paradigm, there is a good probability that the specifications goal will be actually accomplished.

- The volume of the necessary documentation is small compared to the traditional paradigm.

- The paradigm offers a rather detailed guidance to the designer. When the paradigm is mastered, it becomes very natural and easy to use.

However the programming combination also has disadvantages:

- There is a very small possibility of parallel efforts on the project.

- The paradigm is less forgiving than the bottom-up paradigm. Wrong decisions, like wrong decomposition of an item, may make the design unacceptable or competely invalid. The only solution then is to retrace the wrong decision and all consequent decisions, which may be a time-consuming task.

7. Large-small traditional programming.

As remarked earlier, Ada allows the large-small paradigm, where the specifications of the modules are defined first, and then the bodies are defined later. If it turns out that a body of a package is too big, it may be divided into smaller packages by "separate" clauses, and hence this paradigm naturally utilizes the "separate" hierarchy.

In the example in the Appendix, the paradigm requires the specification of the three packages first. Then the bodies can be written in an arbitrary order, or by several programmers in parallel. The compiler will check whether the packages use the items of the other packages correctly, which substantially improves the problems of the integration of the modules into the program. A methodology supporting this paradigm appers in [3].

The problem of this paradigm is its unforgiving nature. Whenever an error in the specification of the packages is made, it tends to affect many additional packages, and a ripple effect propagates through the system. Hence the implementation requires a very careful and detailed design, and the design must be completed before the implementation can start.

Another problem is the high volume of documentation, because the design is a completely separate document than the code. Also because of the unforgiving nature of the paradigm, it is a difficult paradigm.

The advantage is the high parallelism of the implementation effort, where once the design has been completed, many programmers can work on the program in parallel, each implementing a different package.

8. Summary.

In the paper we dealt with the most characteristic combinations of paradigms and life-cycles only. However other combinations are possible: For example, a program can be designed top-down and then implemented bottom-up, or it can be divided into several large packages first (by large-small paradigm) and the packages can be implemented with subpackages by bottom-up paradigm, etc. Each of the combinations displays a different combination of attributes. Hence a manager of a project should carefully evaluate the circumstances of the project, and carefully select the appropriate combination.

As an illustration, let us consider several examples of program implementation:

- An exploratory program is to be implemented by an individual researcher. Because of the exploratory nature of the program, no requirements specifications exist. Hence the appropriate combination is the incremental bottom-up combination, which offers the highest level of forgiveness (and hence if a wrong decision is made, it can be easily corrected), and lowest volume of necessary documentation. The disadvantages of the combination are well compensated for: there is one researcher and hence no need for parallel effort, and the high caliber of the researcher compensates for the higher difficulty of the paradigm.

- An existing program is to be rewitten in Ada by a team of the programmers. The appropriate combination for this situation is the large-small traditional combination. The combination allows several programmers to proceed with implementation in parallel. All disadvantages are reduced by the fact that the old program already exists and is available to the programmers. This is particularly true about the lack of forgiveness, and the high volume of necessary documentation; the experience and the design from the old program can be reused.

- Another program of a well-known application area is to be produced by a small team of programmers. The appropriate combination for this situation is the top-down semi-incremental paradigm. The combination offers the best certainty of compliance with the specifications. The impact of the less forgiving nature of the combination is reduced by the fact that the application area is well-known to the programmers. The impact of the low level of parallelism is reduced by the fact that a small team is involved.

References.

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

[2] Archibald, J.L., Lavenworth, B.M., Power, L.R., Abstract Design and Program Translator: New Tools for Software Design, IBM Systems J. 22, 1983, 170-187.

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

[4] Crew, p., Ward, D., Mungel, G., Analysis of a Prototype Ada Integrated Methodology, Pro. COMPSAC '83 Conf., IEEE Computer Soc., 598-604.

[5] Dijkstra, E.W., The Structure of THE Multiprogramming System, Communications of ACM 11, 1968, 341-346.

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

[7] Kerola, P., Freeman, P., A Comparison of Lifecycle Models, Proc. 5th Intern. Conf. on Software Eng., IEEE/ACM, March 1981, 90-99.

[8] 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.

[9] Lehman, M.M., Programs, Life-Cycles, and Laws of Program Evolution, IEEE Spectrum, Sept. 1980.

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

[11] Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw-Hill, New York, 1983.

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

[13] Rajlich, V., Stepwise Refinement Revisited, to be published, The Journal of Systems and Computers, 1984.

[14] Rajlich, V., A Paradigm for Top-Down Design with Packages, Res. Rep. CLR-TR-31-83, Nov. 1983, Computing Research Lab., 1079 East Engineering, Ann Arbor, MI 48109.

[15] Rajlich, V., SNAP - A Language and Environment for Programming-in-the-Large, to be published in Proc. IEEE Workshop on Languages for Automation, 1984.

[16] Wichman, B.A., Is Ada Too Big? A Designer Answers the Critics, Communications of ACM, Feb. 1984, 98-103.

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

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

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

[20] Zelkowitz, M., Brandstad, M., Proc. ACM SIGSOFT Rapid Prototyping Symposium, Columbia, MD, April 1981.

[21] Wasserman, A.I., Freeman, P., Porcella, M., Characteristics of Software Development Methodologies, in Olle, T.W., Sol, H.G., Tully, C.J., ed., Information Systems Design Methodologies: A Feature Analysis, North Holland, Amsterdam, 1983, 37-62.

Appendix. Eight Queens Problem

The following example was originally published in [17]:

Given are an 8*8 chessboard and 8 queens which are hostile to each other. Find a position for each queen (a configuration) such that no queen may be taken by another queen (i.e. such that every row, column, and diagonal contains at most one queen).

A queen placed on the board attacts the squares as in Fig. 3.

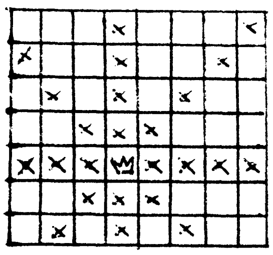

Fig. 3.

Example of a correct solution is in Fig. 4.

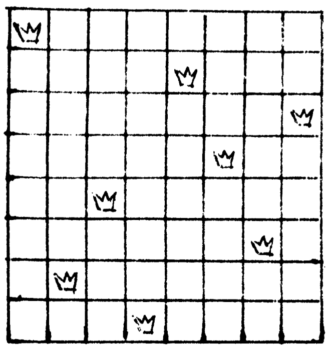

Fig. 4.

The solution to the problem is a program consisting of two packages BOARD and COLUMN and a subprogram MAIN. Package COLUMN defines a type COLUMN and several procedures which deal with that type:

```
package IO_INTEGER is new INTEGER_IO(INTEGER);


with IO_INTEGER; use IO_INTEGER;
package COLUMN is
     type COLUMN is private;
     procedure FIRST(A: out COLUMN);
     function SAFE(A,B: in COLUMN; C,D: in INTEGER) return
                                                 BOOLEAN;
     function LAST(A: in COLUMN) return BOOLEAN;
     procedure NEXT_ROW(A: in out COLUMN);
     procedure PRINT_ROW(A: in COLUMN);
private
     subtype COLUMN is INTEGER range 1..8;
end COLUMN;

package body COLUMN is
     procedure FIRST(A: out COLUMN) is
          begin
               A:=1;
          end FIRST;
     function SAFE(A,B: in COLUMN; C,D: in INTEGER) return
                    BOOLEAN is
          begin
               if A=B or abs(A-B)=abs(C-D) then
                    return TRUE;
               else
                    return FALSE;
               end if;
          end SAFE;
     function LAST(A: in COLUMN) return BOOLEAN is
          begin
               if A=8 then
```

```
                        return TRUE;
                   else
                        return FALSE;
                   end if;
              end LAST;
         procedure NEXT_ROW(A: in out COLUMN) is
              begin
                   A:=A+1;
              end NEXT_ROW;
         procedure PRINT_ROW(A: in COLUMN) is
              begin
                   PUT(A);
              end PRINT_ROW;
    end COLUMN;
```

In package BOARD, data structure BOARD is defined as an array of COLUMN, with the assumption that each column contains one queen only (otherwise the queens attack each other). Also to place 8 queens, it is obvious that each column has to contain exactly one queen.

```
    with COLUMN; use COLUMN;
    package BOARD is
         procedure TRY_FIRST;
         function NOT_FINISHED return BOOLEAN;
         function SUCCESFULL return BOOLEAN;
         procedure ADVANCE;
         procedure REGRESS;
         procedure PRINT_RESULT;
    end BOARD;

    package body BOARD is
         BOARD: array (1..8) of COLUMN;
         I: INTEGER range 1..8;
         procedure TRY_FIRST is
              begin
                   I:=1;
                   FIRST(BOARD(1));
              end TRY_FIRST;
         function NOT_FINISHED return BOOLEAN is
              K: INTEGER range 1..8;
              N: BOOLEAN;
              begin
                   if I<8 then
                     return TRUE;
                   else
                     begin
                       N:=FALSE;
                       for K in 1..7 loop
                         if SAFE(BOARD(K),BOARD(8),K,8) then
                           N:=TRUE;
                         end if;
                       end loop;
                       return N;
```

```
                    end if;
               end NOT_FINISHED;
          function SUCCESFULL return BOOLEAN is
               K: INTEGER range 1..8;
               N: BOOLEAN;
               begin
                    N:=TRUE;
                    for K in 1..I-1 loop
                      if SAFE(BOARD(K),BOARD(I),K,I) then
                        N:=FALSE;
                      end if;
                    end loop;
                    return N;
               end SUCCESFULL;
          procedure ADVANCE is
               begin
                    I:=I+1;
                    FIRST(BOARD(I));
               end ADVANCE;
          procedure REGRESS is
               begin
                    while LAST(BOARD(I)) loop
                      I:=I-1;
                    end loop;
                    NEXT_ROW(BOARD(I));
               end REGRESS;
          procedure PRINT_RESULTS is
               K:INTEGER range 1..8;
               begin
                    for K in 1..8 loop
                      PRINT_ROW(K);
                    end loop;
               end PRINT_POSITION;
     end BOARD;
```

In this package, TRY_FIRST initializes the board to the first position, placing the first queen on the board. Boolean function NOT_FINISHED determines whether we reached the solution. Boolean function SUCCESFULL will determine whether it is possible to place another queen on the board. If yes, we will do so in procedure ADVANCE. If not, then we have to change the current situation, which will be done in REGRESS.

The last compilation unit is a subprogram main:

```
     with BOARD; use BOARD;
     procedure MAIN is
          begin
               TRY_FIRST;
               while NOT_FINISHED loop
                    if SUCCESFULL then
                      ADVANCE;
                    else
                      REGRESS;
                    end if;
          end loop;
          PRINT_RESULTS;
    end MAIN;
```