Stepwise Refinement Revisited

Vaclav Rajlich
University of Michigan

In this paper, rigorous application of stepwise refinement is explored. The steps of definition, decomposition, and completion are described, where completion is a newly introduced step. This combination of steps extends the use of stepwise refinement to larger systems. The notions of range, active objects, and backlog interface are introduced. Verification of incomplete programs via interactive testing is described. The paradigm is demonstrated in an example. The relationship between the paradigm and the current programming languages is considered. It is argued that the WHILE-DO loop is a harmful construct from this point of view.

1. INTRODUCTION

Stepwise refinement is one of the oldest and most widely used methods of program design [3,4,10,11]. Recently, a new interest in stepwise refinement has appeared in connection with software environments, where stepwise refinement is the methodology supported by specialized tools of the environments [1,2,5–9].

The quality of a software design methodology can be characterized by the following interrelated criteria:

1. The generality of the methodology, i.e., the size of the domain of application.
2. The ease of use of the methodology.
3. The consistency of the methodology.

The meanings of the first two criteria are obvious. To explain the third criterion, we assume that a *program design* is a sequence of decisions that lead to a finished program. The role of the methodology is to guide the designer and give advice as to what decision should be made at any particular moment, and on what particular information to base that decision. A methodology is *consistent* if it gives appropriate guidance in all decisions to be made during a program design. Conversely, if it gives poor advice or no advice for some decisions, it is *inconsistent*.

The current stepwise refinement methodology is well suited to the design of small programs; the methodology has found its way into introductory programming texts [3,11]. However, problems arise when larger programs are to be designed by stepwise refinement. Then the methodology becomes difficult to use, and in fact, it becomes inconsistent. The reason is that it is geared solely toward the decomposition of objects (procedures and data). There is at present no organized way to determine the full set of objects to be decomposed. It is assumed that this set is somehow known in advance. This is particularly burdensome in the case of variables, where the programmer is required to determine and declare all of the variables of the program before starting the decomposition process [3,10]. This is only realistic for toy programs. For larger programs, we need a technique different from decomposition, which will help us to determine the set of all objects, and to do this in small increments. In this paper, the technique is called *completion*.

Another problem which has to be addressed is the validation of partial programs. When designing medium-sized systems, the validation cannot be postponed until the system is finished, and hence, it has to be done on partial systems. Various methods have been proposed for such validation [9,13]. However, at present, the most realistic method of validation is by means of testing. A method commonly used is to replace the undefined objects with stubs which approximate the function of those objects. We are suggesting a method called interactive testing, where undefined parts of a program are hand-simulated by the programmer, using stubs to support the simulation. The advantage is that this is a universal method which can be applied in all cases; it makes the methodology explained here consistent.

This paper is divided into four sections. Section 2 contains the basic ideas, Section 3 provides an example, and Section 4 contains a discussion of some language

Address correspondence to Vaclav Rajlich, Department of Computer and Communication Science, University of Michigan, Ann Arbor, MI 48109.

The Journal of Systems and Software 5, 81–88 (1985)
© Elsevier Science Publishing Co., Inc., 1985

0164-1212/85/$3.30

constructs from the point of view of stepwise refinement.

Although the paper is self-contained, familiarity with stepwise refinement as presented in [3,10,11] should be helpful.

2. DEFINITIONS

Programs and software systems consist of *objects* (procedures, functions, variables, etc.), and their *relations* (procedure calls, function calls, access to variables, etc.). Complicated programs consist of many objects and many relations among them. Even a purely mechanical process such as typing of all of the objects in the whole program cannot be done in one stretch; it has to be divided into smaller and more managable steps. The creative process of programming is of course much slower. We must carefully untangle the web of relations among the objects of the target program, and introduce objects and their relations one after the other. During program design, the program 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. Program design is a sequence of incremental steps, each adding something to the existing part and deleting something from the intended part. Throughout the process it is assumed that although the intended part has not yet been written, the designer has a good grasp of its function and its inherent structure.

For the sake of simplicity of explanation, we shall assume that the system consists only of variables, procedures, and functions (they will be called by the generic name "objects"). Also, we will assume that there are only two kinds of relations among the objects. The relation "read" means that the value of a particular variable is being read in a procedure. The relation "write" is the complementary relation.

The *range* of a variable A is the variable A plus all procedures or functions which either read or write the variable A. The range of a procedure P is just the procedure P, and similarly for functions.

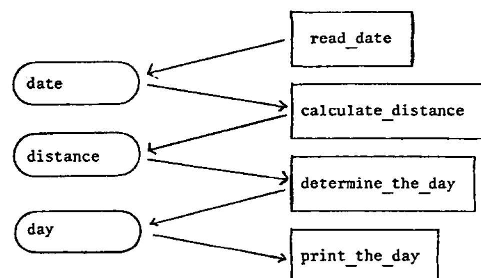
In a typical situation, the existing part contains names of as yet undefined variables, procedures, or functions. For example, the existing part may contain a reference to a procedure "read_data," but the body of the procedure has not yet been defined. We will call objects of this kind *active* objects, and the set of all active objects at a particular time is called the *backlog interface.* The backlog interface is in fact the interface between the existing and the intended parts of the system.

The backlog interfaces are the documentation on which the design decisions of stepwise refinement are based. Although the backlog interface is not a part of the code itself, we recommend that the programmer keep an updated backlog interface at all times, making it a part of the program documentation. In this paper, we assume that the backlog interface is kept in a graphical form, where the active variables are denoted by ovals, active procedures and functions are denoted by rectangles, and the relations "read" and "write" are denoted by arrows, as in Figure 1.

The basic steps of stepwise refinement are *definition* and *decomposition* [10]. *Definition* is a step in which we define an active object in terms 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. An important property of definition is that the smallest unit of definition is a range. Every definition step means that one or several ranges are defined at once. (Another rule governing definition appears in Section 3, step 4.) *Decomposition* is a different step, in which an active object is defined in terms of new active objects. Decomposition and definition are the only steps in stepwise refinement as presented by [10].

There is an alternative way to introduce new active objects during program design. We will call this alternative step *"completion."* In a completion step, we will examine all active objects and try to determine whether they can function correctly, or whether they need to refer to some other objects in order to be able to function. There are two situations which call for the introduction of new objects: first, two procedures may need to communicate with each other, and hence there is a need for a variable which will facilitate this communication. Second, a variable may need an initializing procedure which will allow it to function correctly. This initialization is not a consequence of decomposition, and hence the set of procedures has to be enlarged to

Figure 1. The backlog interface after Step 2. Note that it contains relations "read" and "write" which have not appeared in the previous code. These relations have to be supplied by the programmer, who has an insight into the intended part of the program.



include initialization. An existing part of a program is *complete* when no new objects need 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. Completion steps are conceptually as easy as decomposition and definition, and they extend the methodology to handle medium-size programs, where the ultimate set of objects cannot be predicted in advance. Our methodology requires a completion step after each decomposition step.

In the methodology, we also provide some supporting activities that are helpful in the design process. One of these is an update of the backlog interface. As new active objects are introduced, they are added to the backlog interface, while objects that have been defined are removed from the backlog interface.

It is also useful to keep the list of all relations (based on *read* and *write*) among the active objects, so that the ranges are easily determined. When a new object is introduced, new relations are added to the list; when an object is defined, all of its relations will be removed from the list.

Whenever a program part is complete, it can be tested. The testing is based on the assumption that while no code for the intended part exists, the programmer knows what the functions of the currently active objects are and can hand-simulate their functions. The hand-simulation is supported by stubs which control the interaction between the programmer and the existing part of the program. This is illustrated by the example in the next section.

In summary, the methodology is a sequence of steps described by the following:

```
Introduce the original main program;
REPEAT
  define all objects of one or several ranges
  OR
    BEGIN
      decompose selected objects;
      complete the existing program
    END;
  update the backlog interface;
  interactively test
UNTIL all objects are defined.
```

3. AN EXAMPLE

In this section, we will illustrate the methodology by an example of a program which reads any date of this century (i.e., any date from 1/1/1900 to 12/31/1999), and prints the corresponding day of the week.

We will write the program in an idealized PASCAL-like language. Some comments on current programming languages appear in Section 4.

As a starting step, we will describe the whole program as a call of one procedure:

Step 0.

```
BEGIN
  read_and_calculate_date
END.
```

The procedure is then decomposed:

Step 1.

```
PROCEDURE read_and_calculate_date;
  BEGIN
    Read_date;
    calculate_distance;
    determine_the_day;
    print_the_day;
END.
```

By "calculate_distance" we mean a procedure which determines the number of days between the date processed and a fixed "origin" date. To keep the number small, the distance will always be represented as the total distance MOD 7.

The next step after decomposition is the completion step. We observe that information is passed from read_data to calculate_distance, from calculate_distance to determine_the_day, and from determine_the_day to print_the_day. Hence we need three variables to facilitate the communication. Corresponding to the stepwise refinement philosophy, the question of the types of these variables will be postponed for later consideration:

Step 2.

```
VAR
  date, distance, day;
```

Figure 1 contains the composite backlog after Step 2.

Verification of the program will be done via interactive testing, where *stubs* of active objects will support the interaction with the programmer. The stubs for active data and output data will be the most general type available, i.e., a sufficiently long string of characters, while stubs for procedures will support a dialog with the programmer. When writing the stubs, the backlog interface in Figure 1 is a handy tool.

Step 3.

```
VAR
  date, distance, day, output: ARRAY [1..60] OF CHAR;
PROCEDURE read_date;
  BEGIN
    writeln('Execute read_date.The date is:');
    read(date)
  END;
```

```
PROCEDURE calculate_the_distance;
  BEGIN
    writeln('The date is', date);
    writeln('Execute calculate_the_distance.');
    writeln('The distance is:');
    read(distance)
  END;
PROCEDURE determine_day;
  BEGIN
    writeln('The distance is', distance);
    writeln('Execute determine_day.');
    writeln('The day is:');
    read(day)
  END;
PROCEDURE print_the_day;
  BEGIN
    writeln('The day is:', day);
    writeln('Execure print_the_day.');
    writeln('The output is:')
    read(output)
  END;
```

The program, together with the stubs, would be translated into the programming language we are using (see Section 4), compiled, and executed. Execution would generate the following dialog between the computer and the programmer:

COMPUTER:	Execute read_date. The date is:
PROGRAMMER:	12/5/1984.
COMPUTER:	The date is 12/5/84. Execute calculate_the_distance. The distance is:
PROGRAMMER:	3
COMPUTER:	The distance is 3. Execute determine_day. The day is:
PROGRAMMER:	WEDNESDAY
COMPUTER:	The day is WEDNESDAY. Execute print_the_day. The output is:
PROGRAMMER:	WEDNESDAY
COMPUTER:	(finishes the execution of the program.)

The dialog illustrates the correctness of the existing part of the program.

In the next step, we will define the variables day and distance and their respective ranges.

Step 4.

```
VAR distance:integer;
  day:Array[1..9] OF char;
PROCEDURE determine_day;
  BEGIN
    CASE distance OF 0:day:='Sunday';
                     1:day:='Monday';
                     2:day:='Tuesday';
                     3:day:='Wednesday';
                     4:day:='Thursday';
                     5:day:='Friday';
                     6:day:='Saturday';
    END
  END;
PROCEDURE print_the_day;
  BEGIN
    writeln('The day is', day)
  END;
PROCEDURE calculate_distance;
  BEGIN
    distance_procedure (distance)
  END;
```

Note the way in which the procedure calculate_distance was defined. The procedure is in the range of both the variable distance (defined), and the variable day (undefined), hence it must contain a part which deals with both variables. This part was called distance_procedure. The only way distance_procedure can deal with both defined and undefined variables is to equip it with an actual argument which is the known variable, and keep it in range of the unknown variable. This is a purely mechanical and general step, applicable wherever a procedure (or function) is in range of both undefined and defined variables.

In a similar way, we also deal with functions, where the value to be returned is treated as another argument. The updated backlog interface is illustrated in Figure 2.

For the verification process, we may reuse the stubs of the procedure read_date and the variable date. The procedure distance_procedure has the following stub:

Step 5.

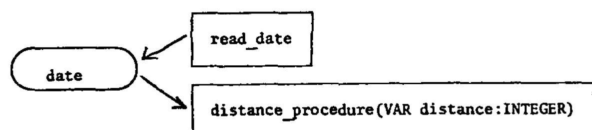
```
PROCEDURE distance_procedure (VAR distance:INTEGER);
  BEGIN
    writeln('Date is', date);
    writeln('Execute distance_procedure');
    writeln('Distance is:');
    read(distance)
  END;
```

The dialog will have a form analogous to the dialog of Step 3. In the next step, we will decompose the variable date and the procedure read_date:

Step 6.

```
VAR date:mm;
         dd;
         zz;
```

Figure 2. The updated backlog interface after step 4.



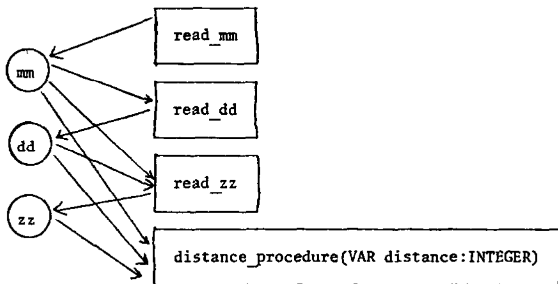


Figure 3. The current backlog interface after step 6.

```
PROCEDURE read_date;
  BEGIN
    read_mm;
    read_dd;
    read_zz;
  END;
```

The program is complete, hence no new objects are obtained by the process of completion. The current backlog interface is illustrated in Figure 3.

Note that distance_procedure has not been decomposed, hence it "inherits" arcs from all components of the former variable date. Also we made an assumption that procedures reads_dd and read_zz will check the correctness of the values read (rejecting dates like 2/30/1982), which created the need of arrows from mm to read_dd and read_zz.

Again, at this moment we may test the program in the style of Step 3.

The next step is the definition of variables dd and zz and their respective ranges. At this moment, we have to decide how robust the program is to be, i.e., what kind of input errors it must be able to recover from. At one extreme, we may declare zz:1900. .1999 which means no robustness at all, because every input error in zz will abort the run of the program. The other extreme is to declare zz:ARRAY[1. .4]OF CHAR, in which case no typing error will cause an abort. A compromise solution chosen here declares zz:INTEGER, where the program will recover from many errors (all incorrect integers), but abort with others (non-numerical symbols).

Step 7.

```
VAR dd,zz:INTEGER;
PROC read_dd;
  BEGIN
    writeln('Enter the day.');
    read(dd);
    WHILE dd < 1 OR dd > month_length DO
      BEGIN
        writln('Incorrect. Enter a different day.');
        read(dd);
      END
  END;
PROCEDURE read zz;
  BEGIN
    writeln('Enter the year.');
    read (zz);
    WHILE(zz < 1900 or zz > 1999)
      OR February AND (dd=29) AND (zz MOD4≠0)
      DO
      BEGIN
        writeln('Incorrect. Enter a different year.');
        read(zz)
      END
  END;
PROCEDURE distance_procedure(VAR distance:INTEGER);
  BEGIN
    distance:=(distance_mm+dd+(zz-1900)+(zz-1901)DIV 4) MOD 7;
    IF (zz MOD 4 = Ø) AND zz ≠ 1900 AND late_month
      THEN distance:=distance+1
  END;
```

The backlog interface after Step 7 (and a completion step) appears in Figure 4. Again, we may test the program with the help of stubs.

The logical step to select now would be to define mm and its range. However, in order to demonstrate the completion step for procedures, let us make a minor detour and decompose distance_mm instead. If we want to rationalize this selection we may argue that at this moment, we are still undecided about the format of mm, the options being:

```
VAR
  mm:INTEGER;
```

and

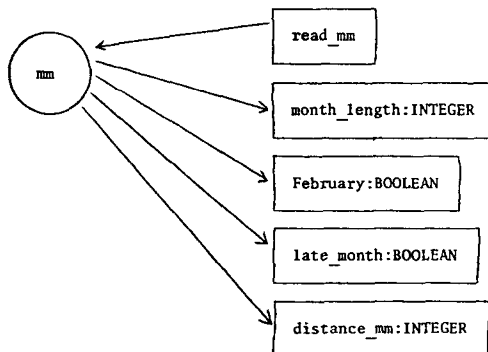
```
VAR
  mm:ARRAY[1 . . 3]OF char.
```

The value of distance_mm will be computed in a loop, in which the lengths of individual months are accumulated.

Step 8.

```
FUNCTION distance_mm:INTEGER;
  BEGIN
    distance_mm:INTEGER;
    WHILE not_over DO
```

Figure 4. The backlog interface after Step 7 (and a completion step).



```
    BEGIN
    distance_:=distance_mm + month_increment;
    next_month
  END;
```

When completing this program, we first notice that not_over, month_increment, and next_month have to communicate through a variable. Let us call this variable

```
VAR
  month;
```

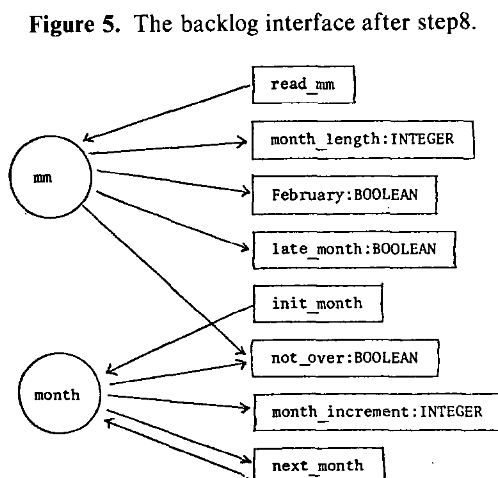
This variable is read in not_over, it is read in month_increment, and it is both read and written in next_month. Tracing the code of function distance_mm, it is obvious that this variable is read before being written, and hence it is not properly initialized. The program cannot work as it has been written, and it must be completed by the appropriate initialization. Let us introduce the procedure init_month, and then function distance_mm will have the following form after the step of completion;

```
CORRECTED FUNCTION distance_mm:INTEGER;
  BEGIN
  distance_:=0;
  init_month;
    WHILE not-over DO
  BEGIN
      distance_mm:=distance_mm + month_increment;
      next_month
    END
  END;
```

The current backlog interface is shown in Figure 5. Again the program can be tested in the style of Step 3.

In the last step, we will define both mm and month and their respective ranges.

Figure 5. The backlog interface after step8.



Step 9.

```
VAR
  mm,month:INTEGER;
PROCEDURE read_mm;
  BEGIN
    writeln('Enter the month.');
    read(mm);
    WHILE mm >12 OR mm < 1 DO
      BEGIN
        writeln('Incorrect. Enter a different month.');
        read(mm)
      END
  END;
FUNCTION month_length:INTEGER;
  BEGIN
    CASE mm OF
      1,3,5,7,8,10,12:month_length:=31;
      4,6,9,11       :month_length:=30;
      2              :month_length:=29
    END
  END;
FUNCTION February:BOOLEAN;
  BEGIN
    IF mm=2 THEN February:=False
    ELSE February:=TRUE
  END;
FUNCTION late_month:BOOLEAN;
  BEGIN
    IF mm>2 THEN late_month:=TRUE
    ELSE late_month:=FALSE
  END;
PROCEDURE init_month;
  BEGIN
    month:=0
  END;
FUNCTION not_over:BOOLEAN;
  BEGIN
    not_over:=month≦mm
  END;
FUNCTION month_increment:INTEGER;
  BEGIN
    CASE month of
      1,3,5,7,8,10,12:=month_increment:=31;
      4,6,9,11       :=month_increment:=30;
      2              :=month_increment:=28
    END
  END;
PROCEDURE next_month;
  BEGIN
    month:=month + 1
  END;
```

4. LANGUAGE CONSIDERATIONS

In Section 3, we used an idealized PASCAL-like language. In general, we are constrained by the real-world languages in which programs are written. In this sec-

tion, we will suggest how to use stepwise refinement in some of the current programming languages. PASCAL is the language which we use as an illustration, but the comments apply to other programming languages as well.

The first, most obvious consideration is that PASCAL does not allow code to be written in the sequence suggested by stepwise refinement. Instead, the programmer has to go back and forth, and he must respect the order of statements of PASCAL with the resulting loss of original clarity and purpose. Some syntax-directed editors [7–9] allow a more flexible order in which statements may be entered, but the program—if printed out—is still organized according to the rules of the original language. We believe that a methodology-oriented program organization has some very important self-documenting properties, and hence this is a considerable loss.

When writing programs by stepwise refinement, procedures and functions of previous sections can either be considered to be closed procedures and functions, or their bodies can be macroexpanded at each occurrence of the call.

Macroexpansion was used in [10,11], and it is considerably better from the point-of-view of the efficiency of the resulting program. However, in the macroexpanded text, the original structure and the original steps are lost, and hence the clarity of the code is substantially diminished.

Moreover, certain PASCAL constructs are not suitable for macroexpansion and require more complicated processing. The most notable example is the WHILE-loop.

Suppose that we have a loop of the form

```
WHILE condition DO body;
```

where condition is a boolean function.

If it decomposes into

```
FUNCTION condition:Boolean;
  BEGIN
    prepare_condition;
    condition: = result_of_preparation
  END;
```

then the resulting text of the loop should be:

```
prepare_conditon;
condition: = result_of_preparation;
WHILE condition DO
  BEGIN
    body;
    prepare_condition;
    condition: = result_of_preparation
  END;
```

As seen in this example, the decomposed body of the function "condition" appears in two places in the new text. This fact may explain why beginning programmers find the WHILE loop so confusing. It also causes considerable difficulty when specialized editors supporting stepwise refinement in PASCAL [8] are implemented.

Note that this problem does not arise in REPEAT-UNTIL loops or in the LOOP-EXIT-END LOOP construct of ADA [12], which are more natural constructs from the point-of-view of stepwise refinement. Of course, it also does not arise when we allow closed functions to be used and do not invoke macroexpansion.

When using closed functions and procedures as the constructs for stepwise refinement, the declarations of variables without types (as in Step 2, Section 3) become meaningless, and are best dealt with as comments in the text. Also note that the organization of the declarations in standard PASCAL leads to an almost complete loss of the methodology-oriented order, with the consequent loss in the clarity of the program.

The reasonable compromise is to combine macroexpansion or textual processing to merge small steps, and deal with larger steps as closed subroutines.

ACKNOWLEDGMENTS

I wish to thank Bernie Galler and Frank Cioch who suggested many improvements in this paper. I would also like to thank Roxianne Carbary for typing this paper.

REFERENCES

1. D. K. Barstow, *Knowledge-Based Program Construction,* North-Holland, New York, 1979.
2. T. E. Cheatham, Jr. G. H. Holloway, and J. A. Townsley, Program refinement by transformation *Proc. 5th Conference on Software Engineering,* San Diego, pp. 430–437.
3. E. B. Coffman, *Problem Solving and Structured Programming in Pascal,* Addison Wesley, 1981.
4. E. W. Dijkstra, Notes on structured programming, in *Structured Programming,* Academic, New York, 1972.
5. A. N. Habermann, An overview of the Gandalf project, Computer Science Research Report 1978–1979, Carnegie-Mellon University, Pittsburgh, 1979.
6. Hans-Ludwig Hansen, and Monica Mullerburg, Conspectus of software engineering environments, in *Tutorial: Software Development Environments,* A. J. Wasserman, ed., IEEE Catalog No. EMO 187-5, pp. 462–476.
7. G. Lyon, Language-Based Editors/Interpreters, *Proc. COMPSAC 82 Conf.,* 1982, pp. 611–612.
8. L. Petrone, A. DiLeva and F. Sirovich, DUAL: An Interactive Tool for Developing Documented Programs by

Step-Wise Refinements *Proc. 6th International Conference on Software Engineering,* IEEE Catalog No. 82CH1795-4, pp. 350–357.
9. T. Teitelbaum and T. Reps, The Cornell program synthesizer: a syntax-directed programming environment, *Commun. ACM* 24, 563–573 (Sept 1982).
10. N. Wirth, Program development by stepwise refinement, *Commun. ACM* 221–227 (April 1971).
11. N. Wirth, *Algorithms + Data Structures = Programs,* Prentice-Hall, 1976.
12. ADA, Military standard MIL-STD-1815, Department of Defense, 1980.
13. R. A. Snowdon, P. Henderson, The TOPD system for computer-aided system development, reprinted in A. I. Wasserman, *Tutorial: Software Development Environments,* pp. 241–262.