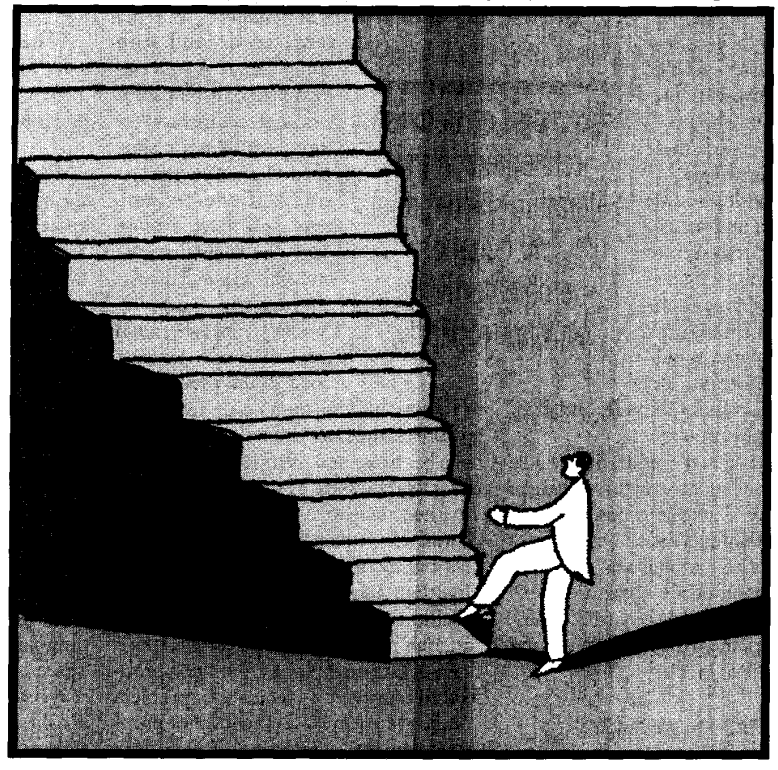
STEPWISE REFINEMENT AND PROBLEM SOLVING

Experiments in the stepwise refinement of pseudocode show that you can express this technique as a combination of problem-solving paradigms.

ROBERT G. REYNOLDS
JONATHAN I. MALETIC
STEPHEN E. PORVIN
Wayne State University



Stepwise refinement remains a popular approach to program-design implementation, primarily because of its flexibility. As originally proposed by Niklaus Wirth, the technique involves successively decomposing design decisions into target code. With each refinement step, the designer must

- note the interdependence of subproblems,
- attempt to preserve the design's functional correctness, and
- defer decisions about representation details as long as possible.[1]

A more specific goal, stated by Harlan Mills, is to "divide, connect, and check an intended function by reexpressing it as an equivalent structure of properly connected subfunctions, each solving part of the problem, and each simpler than the original to further divide, connect and check.... ...each refinement is taken as a working hypotheses for further investigation to be judged either sound or amended as its implication becomes clear."[2]

In this article, we are interested in exploring whether or not the traditional notion of stepwise refinement can be reexpressed as a combination of general problem-solving activities that are based on paradigms taken from artificial intelligence research. This reexpression can form the basis for a more explicit view of programming as a problem-solving activ-

0740-7459/92/0900/0079/$03.00 © IEEE

ity. For intelligent programming environments to become a forceful presence in software development, they must be able to explicitly support the basic problem-solving activities programmers perform.

The examples of stepwise refinement given in the literature seldom refer to these generic problem-solving activities explicitly. We believe it is possible to extract evidence from the stepwise refinement process that will show developers use these activities unknowingly.

In our experiments, we encoded each step of the refinement process into problem-solving activities. We analyzed 26 examples of code implementation using the stepwise refinement of pseudocode. The box on p. 81 lists these examples and gives the author of the problem.

We also developed a model that relates changes in pseudocode structures with generic problem-solving activities.[3] We implemented the model using both refinement and partial metrics to measure changes in pseudocode structure. This technique lets us encode each refinement step in terms of the basic problem-solving activities that it supports.

RELATING METRICS AND PARADIGMS

A pseudocode program has two components, stub and code. Each refinement step is described in terms of the changes made to each component. Designers use two types of metrics, corresponding to stub and code refinement, to quantitatively describe the changes made in each refinement step. Each metric is based on changes in the size of a binary encoded description of its respective component made by a design decision. These encoded descriptions taken together characterize a problem-solving activity.

Each stub can be viewed as a problem to be solved. If more than one stub is dealt with in a refinement step, more than one problem-solving paradigm might be involved because each stub's solution can involve a different problem-solving activity. Therefore, to characterize a refinement step precisely, the designer must know how individual stubs are modified. The refinement activity associated with a single stub is called an *elaboration*.

Changes made in an elaboration are expressed in terms of modifications to the target-code and stub components. A stub can be replaced by new stubs whose structure is either more complex or simpler than the parent stub's. If one or more new stubs have a more complex structure, the stub has expanded relative to some metric. Similarly, an elaboration can either add new target code to the pseudocode or remove it. If the target code's structure is more complex after the elaboration, the code has expanded relative to that metric. If it is less complex, it has reduced.

Examples of stepwise refinement in the literature seldom refer to generic problem-solving activities explicitly.

Thus, stepwise refinement can be viewed as a sequence of elaborations that result in the formation of a program in a target language from an initial function specification. The task of generating a program module given a functional specification is an example of a formation problem. In a formation problem, the problem conditions are given in the form of properties that the solution as a whole must satisfy, and the problem solver has to generate a solution description within a language of solution structures that satisfies the required properties.[4] Here, the problem conditions correspond to the generation of a software system that satisfies a given I/O specification and can be synthesized using a pseudocode language and a target language with a specified grammar.

The steps taken to translate stepwise refinement into a sequence of elaborations, each associated with an activity that supports a paradigm for solving formation problems, are

1. Describe the problem-solving paradigms used to solve formation problems.
2. Predict the changes to pseudocode program structure expected when a problem-solving activity based on a paradigm is performed.
3. Select the metrics that can measure those expected changes.
4. Analyze the results.

DESCRIBING PARADIGMS

The most popular paradigms used to solve formation problems are production system, reduction system, and relaxed reduction.

Production system. This paradigm uses the notion of state-space search. Given the space of all possible pseudocode programs for a pseudocode language and a target language, the problem is to find a sequence of design decisions that produce a path through the state space that terminates with a program in the target language satisfying the required I/O behavior. Herbert Simon describes programming in this way, as a heuristic search through a program space.[5] To express a problem in terms of state-space search, you must define the set of allowable states, the possible moves to go from one state to another, an evaluation function for states, and a state-selection function. There are, in principle, no constraints on allowable moves within the space, so there are no restrictions on the way in which stubs can be elaborated. Problem-solving activities like unrestricted stub decomposition, pruning of the problem space, and backtracking are supported here.

Reduction system. This paradigm is similar to the production-system paradigm, except in the type of allowable moves. The reduction-system paradigm allows only the moves that transform the current state into one closer to completion. In the formation problem described earlier, each stub can be viewed as corresponding to a subproblem to be solved. A problem-reduction move corresponds to the replacement of stubs with new stubs. Each stub is associated with one or more independent subproblems that are easier to solve. A reduction move is said to be terminal if a stub is replaced completely with target code.

The result of any reduction move is to increase the amount of completed code in the program.

Relaxed reduction. This paradigm is used when the subproblems produced by a stub's decomposition are not independent. Relaxed reduction treats each subproblem as independent, solves each separately, and then adjusts the partial solutions to be consistent with each other. Adjustment is either done as soon as inconsistencies arise or deferred until all the subproblems are solved.

PREDICTING CHANGES

The first task in predicting the pseudocode changes expected with each paradigm is to characterize individual elaborations according to how they support the problem-solving activities associated with the paradigms. Table 1 shows a possible classification scheme to associate each problem-solving activity with the changes it is likely to produce. Each category enforces a particular set of constraints on the structural changes made by elaborations that support it. A category may describe an activity associated with more than one paradigm.

Stub reduction. The first row in the table describes activities associated with the two reduction paradigms (reduction system and relaxed reduction). The principle ac-

PROBLEMS USED FOR REFINEMENT SEQUENCES

♦ *Map Color Using Greedy Algorithm.* Construct a greedy to solve a traffic-light design problem as a graph-coloring problem. — A. Aho, J. Hopcroft, and J. Ullmann

♦ *Payroll Calculations.* Compute the federal and state taxes for each employee from tax tables, compute their net income, and generate corporate statistics for all employees. — M. Augenstein and A. Tenenbaum

♦ *Compute X to Power Y:* Compute the function of two integers, x^y, where $x > 1$ and $y \geq 0$. — R.J.R. Back

♦ *Calculate First 1,000 Primes.* Compute the first 1,000 prime numbers in increasing order, starting with 2, and place the ith prime into the ith position of a 1,000-element integer array. — E. Dijkstra

♦ *Calculate First N Primes.* Generate a program to compute the first N prime numbers. — R. Fairley

♦ *Traffic Survey Statistics.* Generate statistics concerning the number of vehicles passing by a vehicle detector over a given period of time. — W. Findlay and D. Watt

♦ *Simulate a Person's Life.* Generate a computer program that simulates a week in the life of an individual. — N. Gehani

♦ *Sum the First N Natural Numbers.* For each element in a list of N positive numbers, compute the sum of all natural numbers up to and including that element.

♦ *Bubble Sort.* Develop a procedure to sort a one-dimensional array of arbitrary size using the bubble-sort algorithm. — P. Gilbert

♦ *Air Pollution Statistics.* Generate average pollution values and reporting errors per hour for a detection device that samples the air every minute over 24 hours. — J. Hughes and J. Mitchton

♦ *Air Pollution Statistics.* (same as above) — R. Jensen and C. Tonies

♦ *Air Pollution Statistics.* (same as above) — R. Linger, H. Mills, and B. Witt

♦ *Exchange Sort.* Read in N integers and arrange them in order from smallest to largest. — D. Ince

♦ *Process Animal Statistics.* Calculate the average weight of pigs in each group of experimented animals as well as the average over groups. — G. Jones and M. Headon

♦ *Calculate Employee Wages.* Compute the take-home pay for an employee including overtime and various deductions, such as medical insurance and taxes.

♦ *Knight's Tour.* Write a program that allows a knight to pass through every square on a chess board only once. — A. Koenig

♦ *External Symbol Table.* Design the matching function for a compiler that searches an external symbol table to see if the name for a token is already present. — G. Myers

♦ *Read and Calculate Days.* Write a program that reads any date from the twentieth century and prints out the corresponding day of the week. — V. Rajlich

♦ *Maximum of Three Numbers.* Compute the maximum of three integers a, b, and c. —J. Denbign Starkey and R. Ross

♦ *Solve Quadratic Equation Roots.* Develop a program to find a value for x in the quadratic equation, $Ax^2 + Bx + C = 0$, where A, B, and C are given as integer values. — D. Watt, B. Wichmann, and W. Findlay

♦ *Sort Names.* Develop a program that reads up to 100 individual names of no more than 20 characters each, and sorts them into lexicographic order.

♦ *Quicksort.* Generate a program that uses the quick-sort algorithm to sort an integer array in ascending order. — J. Welsh, J. Elder, and D. Bustard

♦ *Eight Queens.* Find a position for each of eight queens so that no queen may be taken by another. — N. Wirth

♦ *Text Editor.* Generate a program that supports a set of general text-editing tasks such as the insertion, deletion, and replacement of lines.

♦ *Read and Calculate Date I.* Write a program that, given the year, month, and day of the month, will produce the year, month, and day of the following day, assuming the date lies between January 1, 1900, and December 31, 2099, for the Gregorian calendar. — D. Wood

♦ *Read and Calculate Date II.* (same as above). — D. Wood

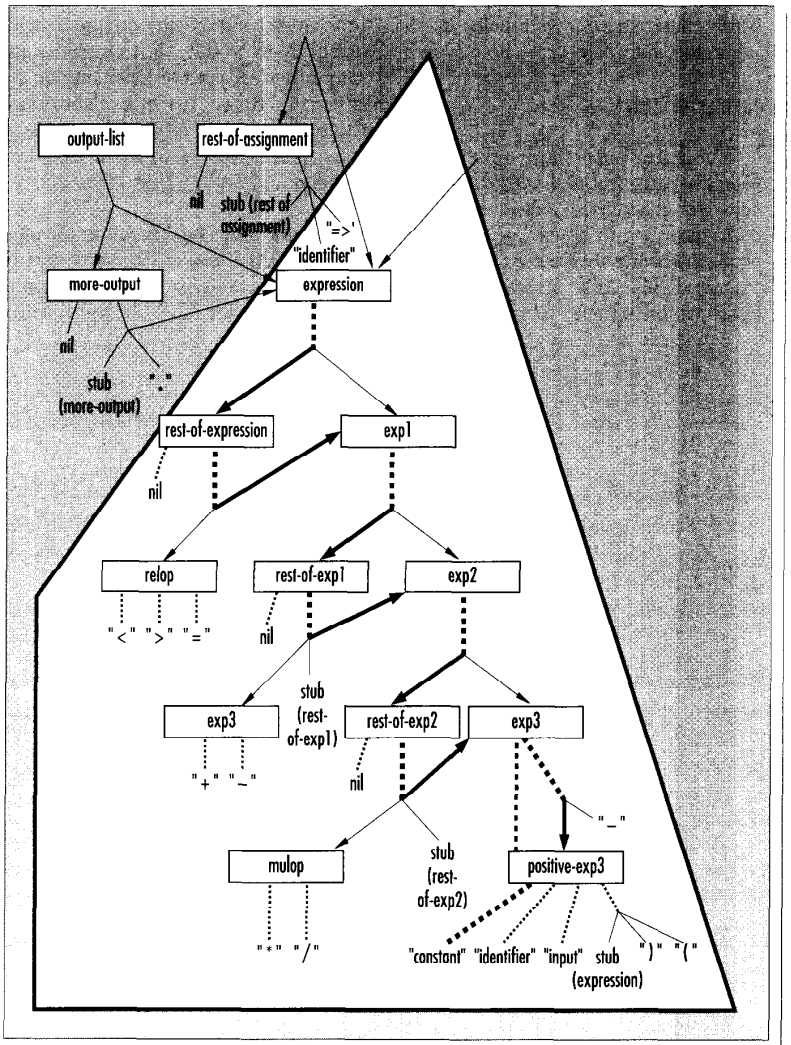


Figure 1. Support subtree and refinement depth and breadth for the expression syntactic class in a directed acyclic graph.

TABLE 1
CLASSIFYING ELABORATIONS ACCORDING TO THE PROBLEM-SOLVING PARADIGM THEY SUPPORT

Changes in stub complexity	Change in code complexity: Code expansion	Change in code complexity: Code reduction
Stub reduction	Problem decomposition (reduction)	Problem interation (relaxed reduction)
Stub expansion	Pruning (solution construction)	Backtracking

tivity in both paradigms is the decomposition of a stub into less complex stubs along with the addition of the target code needed to embed those stubs. This corresponds to a combination of stub reduction and code expansion. Problem decomposition falls in the first row, second column of the table. All the paradigms can support this activity to different degrees.

If the subproblems generated by decomposition are not independent, the designer may have to adjust the partial solutions to reflect these dependencies. When the adjustments made during stub decomposition allow for the consolidation or reduction of existing code structures, stub reduction and code expansion are exhibited together. This resolution activity falls in the first row, third column. The strategy behind relaxed reduction is to solve each subproblem separately. Thus, adjustment often removes redundant code in the solution of each subproblem rather than adding new code.

Stub expansion. The second row of the table shows stub-expansion activities. These are the general activities that support problem solving. The second column describes elaborations that produce both stub and code expansion. Pruning of structural solutions falls into this category. In pruning, a stub of a given syntax class is replaced with a stub of an equal or more complex class. The new stub is embedded in new target code that reflects a more precise interpretation of a stub's characterization, pruning away old alternatives.

For example, a stub of class statement is replaced by target code representing a conditional branch and a new stub of class statement, embedded within the code. In this example, a specific syntactic structure was selected, removing other alternatives from consideration.

Backtracking, which corresponds to stub expansion and code reduction (second row, third column) is also an important activity in problem-solving systems. Decomposition can lead to an inadequate solution, so designers backtrack to an earlier point in the design by removing portions of existing code and replacing the associated stubs with more complex ones.

Backtracking activities will always be a necessity in the generation of novel designs.

SELECTING METRICS

Designers use two types of metrics to describe changes in the pseudocode. Refinement metrics are used to track changes in the stub component;[6] partial metrics are used to track changes in the code component.[7]

Refinement metrics. Refinement metrics characterize the difficulty of the implementation task associated with the decomposition of a stub into target code. The difficulty is expressed in terms of the grammar for the target language. The estimated complexity of the task is computed relative to the nonrecursive directed acyclic graph for the grammar. That is, all indirect and direct recursion is eliminated from the productions that constitute the grammar. Figure 1 shows part of the constrained graph for the Backus Naur form grammar of a simple language. Each nonterminal node represents a syntactic class or category and has an associated support subtree consisting of the set of all possible nodes and arcs reachable from the nonterminal node.

Refinement depth represents the number of decisions necessary in the worst case to turn a stub associated with a syntactic class into complete code. Depth is determined by counting each arc in the longest path in a syntactic class's support subtree from a root to a leaf. In Figure 1, the support subtree for the syntactic class is outlined in a triangle-like shape. The longest path is delineated by bold arcs. It takes a maximum of eight productions to transform the expression into complete target code.

Refinement breadth corresponds to the diameter for the subtree associated with a syntactic class. It is measured by counting the number of unique productions in the class's support subtree. In the same figure, the 22 unique productions are shown in dashed lines.

Refinement volume reflects the worst-case number of bits needed to encode (describe) the sequence of productions used in implementing a syntactic class associated with a nonterminal node, *nt*. It is calculated by combining refinement depth and breadth:

$$\text{volume}(nt) = \text{depth}(nt) \times \log2(\text{breadth}(nt))$$

where log2(breadth) is the number of bits required to uniquely encode any of the nonterminal nodes reachable from *nt* at each step and depth is the worst-case number of steps needed to implement the class.

Refinement volume measures changes in the structural complexity of a stub produced during elaboration.[6] It measures the binary encoding of the production sequences (from the target language's grammar) needed to transform the syntactic category associated with the stub into target code. As such it measures how difficult in terms of language, the stub is to implement. Stub reduction occurs when the refinement volume associated with each new stub is less than the size of the implementation task associated with its parent. Stub expansion occurs when the refinement volume for at least one of the new stubs is greater than that of its parent.

Partial metrics. Partial metrics, which describe code complexity, are based on the idea that pseudocode has two component classes: projected and prescribed.[7] The projected component consists of the current set of stubs in the program. The information associated with these stubs is used to make projections about their contributions to the overall structural complexity of the completed code. The prescribed component consists of all the target code currently in the pseudocode program.

For example, Halstead's Volume[8] measures the size of a program in a target language in terms of the number of bits needed to encode it. For a pseudocode program, you can compute Halstead's Volume by ignoring the contribution of the projected or stub component. This is called the prescribed volume. Its precise formulation is **N**log2**n**, where **N** is the total number of operator and operand occurrences in the program and **n** is the number of unique occurrences of both operators and operands.

Changes in the target code's structural complexity produced during the elaboration of a stub are expressed as changes in prescribed volume. Code expands when the prescribed volume for the pseudocode program increases after elaboration. Code reduces when the prescribed volume decreases after elaboration.

Refinement metrics characterize the difficulty of the implementation task associated with the decomposition of a stub into target code.

ANALYZING RESULTS

The last step in relating stepwise refinement to problem solving is to translate the elaboration classifications into the problem solving activity they represent. In all 26 problems, we computed the change in refinement volume and prescribed volume for each elaboration. We then used this data to encode stepwise refinement in terms of the problem-solving activities using the classification scheme in Table 1.

Table 2 gives the number of elaborations in the sample that correspond to each problem-solving operation. All problem-solving categories are represented in the examples.

The table shows that elaborations performing problem decomposition are by far the most frequent, but they account for less than 90 percent of all elaborations. The next most frequently occurring activity is backtracking. The 23 observed instances occurred in 10 of the 26 refinement examples, which indicates that some problems seem to require more rethinking of the solution than others. Problem decomposition and backtracking together constitute approximately 92 percent of the elaborations.

The next most frequently observed activity, relaxed reduction, occurs when existing code is reduced in conjunction with a stub-reduction decision. These transformations represent code-optimization decisions for the most part. Relaxed-reduc-

tion activities that produce more complex target code in conjunction with stub reduction are counted in the problem-reduction class.

The final category is pruning. Only six elaborations, from four refinement examples, are of this type. Although the numbers for this activity are rather small, it is still important in the overall process. The problems that use pruning are some of the more complex among the 26. These examples may actually be more representative of the real-world problems designers face. That these operations are present, even as part of a solution for a relatively simple problem, indicates their importance.

SAMPLE SEQUENCES

To illustrate the variety of paradigms we actually observed in the 26 examples, we present three encoded sample refinement sequences. (We modified the problems slightly to help standardize data collection and presentation.) Each example is described as an indexed sequence of refinement steps. A refinement step can consist of multiple elaborations, and each elaboration is encoded to represent one of the four categories in Table 1. Each refinement step is described as a collection of symbols, in which each symbol corresponds to an elaboration class.

TABLE 2
332 ELABORATIONS CLASSIFIED ACCORDING TO BASIC PROBLEM-SOLVING STRATEGIES

Changes in stub complexity	Change in code complexity: Code expansion	Change in code complexity: Code reduction
Stub reduction	292 Problem reduction	11 Relaxed reduction
Stub expansion	6 Pruning	23 Backtracking

Eight Queens. The example in Figure 2 is the solution to the Eight Queens problem (described in the box on p. 81) as described by Wirth in his ground-breaking paper on stepwise refinement.[1] The encoded sequence demonstrates a consistent problem-reduction approach over the first 13 refinement steps, as Table 3 shows.

The final refinement consists of two major activities. One is performed by an elaboration that supports a relaxed-reduction approach, which led to a more efficient code structure. The other is the selection of an approach for handling a termination process that emerged during the finishing touches of the design. The new stub is more complex than its parent because it contains a more specific solution structure.

Because the design in the Eight Queens example focuses on implementing a particular approach (algorithm or heuristic), problem reduction is the dominant activity followed by a set of activities that tie up loose ends in the design. Tying up loose ends is similar to making design decisions that depend on decisions about other tasks. Examples of tasks that often exhibit such interdependence are code optimization and exception handling, both of which are in the Eight Queens example.

TABLE 3
EXCERPTS OF THREE ENCODED REFINEMENT SEQUENCES

Refinement number	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Wirth Eight Queens	PR	PR PR	PR PR PR	PR	PR	PR	PR	PR	PR	PR	PR	PR	PR	RR P
Starkey and Ross Maximum of 3 Numbers	PR	PR	RR	PR	PR	B	RR	PR						
Rajlich Read and Calculate Days	PR	PR PR PR PR	B PR PR	PR	B	B PR PR	PR	PR PR PR PR PR PR PR PR						

PR is problem reduction, RR is relaxed reduction, P is pruning, and B is backtracking.

```
Refinement 0: Stub: EightQueens
  BOARD;
  POINTER;
  SAFE;
  CONSIDERFIRSTCOLUMN;
  loop
    TRYCOLUMN;
    if SAFE then
      SETQUEEN;
      CONSIDERNEXTCOLUMN;
    else
      REGRESS ;
    end if ;
    exit when LASTCOLDONE
      or REGRESOUT1STCOL;
  end loop;
Refinement 1a: Stub: TRYCOLUMN
  procedure TRYCOLUMN is
    begin
      loop
        ADVANCEPOINTER ;
        TESTSQUARE ;
      exit when SAFE or LASTSQUARE ;
      end loop ;
    end ;
Refinement 1b: Stub: REGRESS
  procedure REGRESS is
    begin
      RECONSPRIORCOLUMN ;
      if not REGRESOUT1STCOL then
        REMOVEQUEEN ;
        if LASTSQUARE then
     RECONSPRIORCOLUMN ;
          if not REGRESOUT1STCOL
          then
           REMOVEQUEEN ;
        end if ;
        end if ;
      end if ;
    end ;
Refinement 2a: Stub: BOARD
  BOARD : array ( 1 .. 8 ) of INTEGER ;
Refinement 2b: Stub: POINTER
  POINTER : INTEGER ;
Refinement 2c: Stub:SAFE
  SAFE : BOOLEAN;
Refinement 3: Stub:CONSIDERFIRSTCOLUMN
  procedure CONSIDER_1ST_COLUMN is
    begin
      J := 1 ;
      X(J) := 0 ;
    end ;
Refinement 4: Stub: CONSIDERNEXTCOLUMN
  procedure CONSIDERNEXTCOLUMN is
    begin
      J := J + 1 ;
      X(J) := 0 ;
    end ;
Refinement 5: Stub: RECONSPRIORCOLUMN
  procedure RECONSPRIORCOLUMN is
    begin
      J := J - 1 ;
    end ;
Refinement 6: Stub: ADVANCEPOINTER
  procedure ADVANCEPOINTER is
    begin
      X(J) := X(J) + 1 ;
    end ;
Refinement 7: Stub: LASTSQUARE
  function LASTSQUARE return BOOLEAN is
    begin
      LASTSQUARE := X(J) = 8 ;
    end ;
Refinement 8: Stub: LASTCOLDONE
  function LASTCOLDONE return BOOLEAN is
    begin
      LASTCOLDONE := J > 8 ;
    end ;
Refinement 9: Stub: REGRESOUT1STCOL
  function REGRESOUT1STCOL return
    BOOLEAN is
    begin
      REGRESOUT1STCOL := J  1 ;
    end ;
Refinement 10: Stub: TESTSQUARE
  A    : array (  1 ..  8 )  of BOOLEAN;
  B    : array (  2 .. 16 ) of BOOLEAN;
  C    : array ( - 7 ..  7 ) of BOOLEAN;
    procedure TESTSQUARE is
    begin
    SAFE := A (X(J)) AND  B(J + X(J))
            AND  C(J - X(J)) ;
    end ;
Refinement 11: Stub: SETQUEEN
  procedure SETQUEEN is
    begin
      A ( X(J) )     := false ;
      B ( J + X(J) ) := false ;
      X ( J - X(J) ) := false ;
    end ;
Refinement 12: Stub: REMOVEQUEEN
  procedure REMOVEQUEEN is
    begin
      A ( X(J) )     := true ;
      B ( J + X(J) ) := true ;
      C ( J - X(J) ) := true ;
    end ;
Refinement 13a: Stub: OUTPUT_RESULTS;
  if J > 8 then PUT ( X ) ;
    else FAILURE;

Refinements 13b through 13h: In ConsiderNextColumn, add
code X(J) := I; in all procedures make the following refinement:
replace X(J) by I
```

Figure 2. *Refinement sequence for Wirth's Eight Queens problem.*

```
Refinement 0: Stub: MAX_OF_3
  procedure MAX_OF_3 is
    A, B, C : INTEGER;
    begin
      GET ( A, B, C );
      PUT (MAX  ( A, B, C ) );
    end ;

Refinement 1: Stub: MAX
  function MAX ( A , B , C : in INTEGER )
    return INTEGER is
    begin
      if A > B then
        if B > C then
          MAX := A;
        else
          MAX := C;
        end if;
      else
        if B > C then
          MAX := B;
        else
          MAX := C;
        end if;
      end if;
    end;

Refinement 2: All prescribed code in function MAX is
changed to
    LARGE : INTEGER;
    if A > B  then
      LARGE := A;
    else
      LARGE := B;
    end if;
    if LARGE > C  then
      MAX := LARGE ;
    else
      MAX := C ;
    end if;

Refinement 3: All prescribed code in refinement 2 is changed to
    if (A > B) and (A > C) then
      MAX := A;
    end if;
    if (B > C) and (B > A) then
      MAX := B;
    end if;
    if (C > B) and (C > A) then
      MAX := C;
    end if;

Refinement 4: In function MAX change > into a >=.

Refinement 5: All prescribed code in refinement 3,4 is changed
to
    LARGE : INTEGER;

    LARGE := MAXIMUM (A, B);
    LARGE := MAXIMUM (LARGE, C);
    MAX := LARGE ;

Refinement 6: All prescribed code in refinement 5 is changed to
    MAX := MAXIMUM (MAXIMUM (A, B),C);

Refinement 7: Stub:MAXIMUM
  function MAXIMUM ( A , B  : in INTEGER )
    return INTEGER is
    begin
      if A > B then
        MAXIMUM := A;
      else
        MAXIMUM := B;
      end if;
    end;
```

Figure 3. Refinement sequence for Starkey and Ross's Maximum of Three Numbers problem.

This sequence of problem-reduction operations followed by the solution of subproblems requiring global information is certainly the norm among the 26 examples. Designers can use this approach when the problem is well-defined and they know the algorithmic structure of its solution for the most part.

Some refinement sequences in other examples differed markedly from this approach primarily because the nature of the problems to be solved was different. Two possible alternative problem classes are

1. Problems that can be solved in a transformational manner. That is, an initial unsophisticated prototype is produced, and then incrementally transformed into a more complex version that satisfies given constraints. Some of these transformations may not preserve correctness and may require backtracking

2. Problems whose solution (algorithm/heuristic) cannot be completely known in advance and must be modified during design. The solution may be a collection of special cases or rules, in which not all the relevant rules are known ahead of time.

The problem-solving activities used to solve these alternative problem classes may also differ significantly from the norm. An encoded refinement sequence of each is given in Table 3 along with the Eight Queens problem.

Maximum of Three Numbers. Figure 3 shows this example, which reflects the transformational approach. The task is to compute the maximum of three numbers. The initial approach is a simple extension of a solution that works for two numbers. This is implemented as two problem-reduction steps and a third relaxed-reduction step.

In step 3, the conditional control is reworked. This function did not produce the desired I/O behavior and was modified by the transformation in step 4.

```
Refinement 5c: Replace prescribed code from within procedure
in reference with the following prescribed code and stubs:
  DISTANCE := (DISTANCE_MM + (YY – 1900)
       + (YY – 1901) DIV 4) MOD 7;
  if (YY MOD 4 = 0) AND YY /= 1900
     AND LATE_MONTH then
     DISTANCE := DISTANCE + 1;
  end if;

Refinement 6: Stub: DISTANCE_MM
function DISTANCE_MM return INTEGER is
  begin
    DISTANCE := 0;
    INIT_MONTH;
    while NOT_OVER loop
      DISTANCE_MM := DISTANCE_MM +
        MONTH_INCREMENT;
      NEXT_MONTH;
    end loop;
  end DISTANCE_MM;
(Also change DISTANCE_MM to prescribed function call.)

Refinement 7a: Stub: READ_MM;
  MM : INTEGER;
    procedure READ_MM is
    begin
      PUT (" ENTER THE MONTH : ");
      GET (MM);
      while (MM > 12  OR MM  1) loop
        PUT ("INCORRECT!
         ENTER DIFFERENT MONTH ");
        GET (MM);
      end loop;
    end READ_MM;
(Also change READ_MM to prescribed procedure call.)

Refinement 7b: Stub: MONTH_LENGTH
    function MONTH_LENGTH return INTEGER is
  begin
    case MM is
    when 1,3,5,7,8,10,12
      => MONTH_LENGTH := 31;
    when 4 , 6 , 9 , 11
      => MONTH_LENGTH := 30;
    when 2 => MONTH_LENGTH := 29;
    end case;
  end MONTH_LENGTH;
(Also change MONTH_LENGTH to prescribed function call.)

Refinement 7c: Stub: FEBRUARY
function FEBRUARY return BOOLEAN is
  begin
    if MM = 2 then
      FEBRUARY := FALSE;
    else
      FEBRUARY := TRUE;
    end if;
  end FEBRUARY;
(Also change FEBRUARY to prescribed function call.)

Refinement 7d: Stub: LATE_MONTH
function LATE_MONTH return BOOLEAN is
  begin
    if MM > 2 then
      LATE_MONTH := TRUE;
    else
      LATE_MONTH := FALSE;
    end if;
  end LATE_MONTH;
(Also change LATE_MONTH to prescribed function call.)

Refinement 7e: Stub: INIT_MONTH
procedure INIT_MONTH is
    begin
      MONTH := 0;
    end INIT_MONTH;
(Also change INIT_MONTH; to prescribed procedure call.)

Refinement 7f: Stub: NOT_OVER
  function NOT_OVER return BOOLEAN is
    begin
      NOT_OVER := MONTH  MM;
    end NOT_OVER;
(Also change NOT_OVER to prescribed function call.)

Refinement 7g: Stub: MONTH_INCREMENT
function MONTH_INCREMENT return BOOLEAN is
  begin
    case MONTH is
    when 1,3,5,7,8,10,12
      => MONTH_INCREMENT := 31;
    when 4,6,9,11
      => MONTH_INCREMENT := 30;
    when 2 => MONTH_INCREMENT := 28;
    end case;
  end MONTH_INCREMENT;
(Also change MONTH_INCREMENT to prescribed function
call.)

Refinement 7h: Stub: NEXT_MONTH
procedure NEXT_MONTH is
    begin
      MONTH := MONTH + 1;
    end NEXT_MONTH;
```

Figure 4. *Refinement sequence for Rajlich's Read and Calculate Day problem.*

Still the control complexity was too high, so the designer backtracks by removing the detailed code and replacing it with a simple function call in step 5. Step 6 corresponds to reducing the two calls into one and is an example of relaxed reduction. The final problem-reduction step completes the process.

In this example, the problem-solving activities combine to produce a simple rapid-prototyping solution within the context of stepwise refinement.

Read and Calculate Day. Figure 4 shows this program, which reads any date in the twentieth century and prints out the day of the week. The solution is effectively a collection of transformation rules. Initially, some of the more frequently applied rules are known but new situations develop as the design proceeds. These situations often relate to exception handling. As a result, backtracking takes place frequently as new situations are discovered and added to the design. This situation is particularly relevant with somewhat fuzzy problems like interface design.

In problems such as this one, the goal is to satisfy the user, so the initial functional specification doesn't have enough information for the designer to structure a solution directly. Thus, the designer uses a combination of problem-solving strategies that incrementally augment the current solution.

To develop a useful, intelligent programming environment, designers must understand the nature of the problem-solving activities to be performed in that environment. Our experiments with a variety of problems show that designers can perform several problem-solving activities within the stepwise-refinement framework. The presence of certain combinations of activities suggests that programmers are implicitly emulating certain paradigms that have proved useful in solving complex problems. Also, as the Wirth example suggests, a particular paradigm and its associated activities seem to be applied often throughout the refinement sequence for a given problem. It is also clear that the nature of the problem to be solved influences the type of activities performed to achieve a solution, as well as the problem-solving paradigm that they implicitly support.

Whether the nature and the frequency of these problem-solving activities for textbook examples carries directly over to real-world activities remains to be seen. For example, backtracking and pruning will occur more often when a designer is faced with more sophisticated problems.

We hope to see problem-solving paradigms and their supporting activities more explicitly integrated into stepwise refinement. We also expect to see new problem-solving paradigms emerge that are unique to software development. ♦

REFERENCES

1. N. Wirth, "Program Development By Stepwise Refinement," *Comm. ACM*, Apr. 1971, pp. 221-227.
2. R. Linger, H. Mills, and B. Witt, *Structured Programming Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
3. S. Porvin, R. Reynolds, and J. Maletic, "An Empirical Study of the Use of Problem Reduction as a Paradigm for Problem Solving in Software Engineering," *Proc. ACM Computer Science Conf.*, ACM Press, New York, 1991, pp. 618-629.
4. S. Amarel, "Problem Solving" in *Encyclopedia of Artificial Intelligence: Volume 2*, S. Shapiro, ed., John Wiley & Sons, New York, 1990, pp.767-779.
5. H. Simon, "Whether Software Engineering Needs to be Artificially Intelligent," *IEEE Trans. on Software Eng.*, July 1986, pp. 726-732.
6. R. Reynolds and J. Maletic, "An Introduction to Refinement Metrics: Assessing a Programming Language's Support of the Stepwise Refinement Process," *Proc. ACM Computer Science Conf.*, ACM Press, New York, 1990., pp. 82-88.
7. R. Reynolds, "The Partial Metrics System: Modeling the Stepwise Refinement Process Using Partial Metrics," *Comm. ACM*, Nov. 1987, pp. 956-963.
8. M. Halstead, *Elements of Software Science*, Elsevier North Holland, New York, 1977.

Robert G. Reynolds is an associate professor of computer science at Wayne State University, where his main interests are intelligent programming environments, the acquisition of software-engineering knowledge, genetic algorithms, and adaptive systems. He is also head of the university's PM project, which deals with the use of machine learning tools in the acquisition of software-engineering knowledge. He has authored or coauthored more than 60 papers and a book.

Reynolds holds a PhD in computer science from the University of Michigan with a specialization in artificial intelligence (machine learning). He is a member of the IEEE Computer Society, ACM, and AAAI.

Jonathan I. Maletic is a PhD candidate in computer science at Wayne State University and a doctoral student working in artificial intelligence. His particular interests are automated software design, machine learning, software reuse, and reverse engineering.

Maletic holds a BS in computer science from the University of Michigan at Flint and an MS in computer science from Wayne State University He is a student member of the IEEE Computer Society, ACM, and AAAI.

Stephen E. Porvin is a senior knowledge engineer at Inference Corp., where his interests include knowledge engineering, expert systems, and the application of artificial intelligence to software engineering.

Porvin holds a BA in anthropology from the University of Michigan and a BS and an MS in computer science from Wayne State University. He is a member of the IEEE Computer Society, ACM, ACM's SIGArt and AAAI.

Address questions about this article to Reynolds or Maletic at Wayne State University, CS Dept., 431 State Hall, Detroit, MI 48202; Internet rgr@cs.wayne.edu and jlm@cs.wayne.edu.