Improving Undergraduate Students' Programming Skills

Sukhamay Kundu
Computer Science Department, Louisiana State University
Baton Rouge, LA 70803, USA
e-mail: kundu@csc.lsu.edu

Abstract – Two key factors for the failure of undergraduate students in creating high quality programs are their inability to: (1) identify the basic steps in building a high-level solution algorithm, and (2) convert these steps into an elegant and efficient program implementation. Both these tasks require creative thinking and a systematic approach with emphasis on the programming process. We believe that with proper training the students can improve their programming skills and create high-quality programs. We describe a new approach to programming, which can be viewed as a refinement of the well-known "stepwise refinement" method. We use the notion of work-breakdown structure to address factor (1) and suggest a few coding techniques to address factor (2). Our initial experience in using the new approach has been very positive in terms of improved quality of student programs.

Keywords: work-breakdown structure; stepwise refinement; programming process.

I. INTRODUCTION

The low success rate in the first and second year undergraduate computer programming courses continues to be a serious problem even today after the decades of many advances in Programming Language designs (including object-oriented languages) and Software Engineering methods. A drop-out rate of 30% in the first year programming courses is not uncommon. We often see another 20-30% of the remaining students having difficulties with programming. This means 45-50% of the incoming computer science undergraduates either change their major or struggle with their computer science degree program. Advances in the programming methods from structured programming and stepwise refinement to object-oriented (OO) programming has not led to significant improvements in the undergraduate students' programming ability [2]. For some students, the problem persists beyond their undergraduate years.

The key element of our new approach to programming is the use of work-breakdown structure (WBS) in building the high-level pseudocode or algorithm. The notion of WBS is routinely used by business analysts as a tool in planning and scheduling. The connection of WBS to the stepwise refinement method (SRM) of program development is that both use a top-down approach, where a task is successively decomposed into disjoint subtasks until we arrive at subtasks that can be readily solved. Because WBS is not concerned with the control-logic (if-then-else and loops), it is simpler than SRM where one develops subtask decomposition in parallel with the refinement of control-logic in the form pseudocode. This makes WBS a good intermediate step and one can add the control-logic later in a bottom-up or top-down fashion at each subtask-decomposition step. We can say that WBS helps to operationalize SRM in the same way that SCRUM software development method helps to operationalize agile programming. Both SRM and WBS can accommodate refinements of data, operation, and control-logic. There have been many advances in SRM-based formal techniques for program development [5-7], but these techniques are not suitable for teaching undergraduate-level programming.

We limit ourselves here to non-OO programming. Indeed, one cannot create a high quality OO-program, which involves the added complexity of class-subclass considerations, if one cannot create a small high-quality non-OO program. For the present discussion, a high quality program means simple and clear logic in the basic algorithm, clean implementations for good computational efficiency and memory usage, and simple and clear input/output interfaces.

A. Work-Breakdown Structure

We build a WBS in a top-down fashion and represent it as a tree, where the children of a node shows the decomposition of the node into two or more subtasks. The terminal nodes, called the *work-units*, represent the actual tasks performed in solving the original problem given by the root of WBS-tree. The intermediate nodes represent a hierarchical grouping of work-units into larger conceptual "chunks" and they help us in arriving at the work-units. Creating a WBS-tree is often a non-trivial task and requires much thought and insight into the problem at hand. Although, in principle, a node with $n \geq 3$ children can be replaced by a chain of $n - 1$ nodes each with 2 children, this is not always easily done and nor it is advisable

 ISBN: 978-1-61208-230-1

to force this because it can lead to artificial subtasks. A WBS represents a solution approach in a more abstract form than a pseudocode because it does not show the control-logic. The simultaneous development of control-logic and the bodies of loops and the then/else parts of if-statements makes SRM harder to apply than our two-step method: build a WBS first and then add the control-logic.

B. Two Demonstration Problems

We consider two problems to show the usefulness of WBS in developing complex pseudocodes and their efficient implementations. The solution of the first problem, the triangle classification (TRC), involves a complex nested if-then-else and the solution of the second problem, the constrained binary string generation (CBSG), involves several interacting pairs of loops and if-statements. In both cases, we want simplest logic and the maximum efficiency. The use of WBS helps to create a proper high-level design for the solution algorithm, and then a proper choice of data-structures leads to an efficient implementation. The importance of the CBSG problem is that many other problems can be either formulated in this form or can be solved by generalizing or modifying the method used for solving CBSG. For example, we can associate the subsets of size m of a set of size n with the binary strings of length n with m ones and thus we can generate those subsets using the solution of the CBSG problem. As another example, we can generate all permutations of $n \geq 2$ items by modifying the method for solving CBSG.

Sections II and III present the solutions of the TRC-problem and the CBSG-problem. Some important coding techniques are described in Section IV. Section V gives the conclusion and the future work.

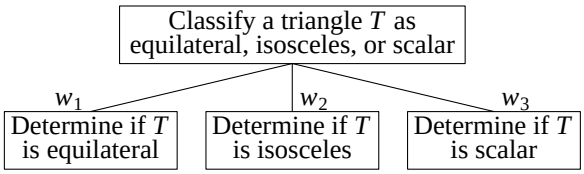
II. TRC-PROBLEM

Here, we want to classify a triangle T as one of equilateral, isosceles, and scalar given the lengths a, b, and c of its sides which are assumed to satisfy the triangle inequalities: $a + b > c$, $a + c > b$, and $b + c > a$. Table 1 shows the conditions for classifying T. Note that we don't need the condition "$a = c$" in C_1 and likewise the condition "$b \neq c$" in the first and-combination of C_2, etc.

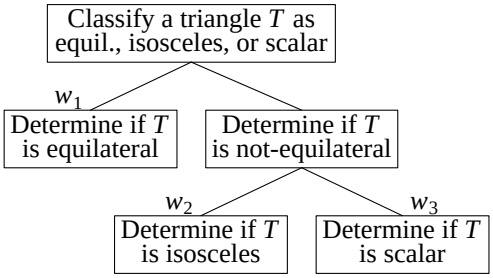
Table 1. Conditions for triangle classification.

Type	Condition
Equilateral	C_1: $(a = b) \wedge (b = c)$
Isosceles	C_2: $[(a = b) \wedge (a \neq c)] \vee [(a = c) \wedge (a \neq b)] \vee [(b = c) \wedge (a \neq b)]$
Scalar	C_3: $(a \neq b) \wedge (a \neq c) \wedge (b \neq c)$

Figures 1(i)-(ii) show two WBSs for the TRC-problem, each with three work-units w_1 to w_3. There are two more WBSs similar to Figure 1(ii) based on decomposing the root with respect to isosceles triangle and scalar triangle. Clearly, the WBSs for any classification problem can be mapped in an one-to-one fashion to the classification trees for that problem if we do not include additional computation details in the WBSs for the classification problem.



(i) A simple WBS.



(ii) A multi-level WBS with more structure.

Figure 1. Two alternate WBS for the TRC-problem.

Figure 2 shows several alternate pseudocodes for the TRC-problem. The one in Figure 2(i) comes directly from the WBS in Figure 1(i) and is the least efficient; it does not take advantage of the disjointness of conditions C_1-C_3. The pseudocode in Figure 2(ii) is from [3]; it is little better than the one in Figure 2(i) but it fails to fully exploit the relationships among the equalities and inequalities in C_1-C_3. Figure 2(iii) is obtained from Figure 2(ii) by replacing C_3 by its negation (thereby avoiding negations like "≠") and interchanging the associated then-else parts; this is better than using C_2 in terms of the number of evaluations of various "=" and "≠". Figure 2(iv) gives the most efficient and elegant solution. In terms of E = the average number of evaluations of "=" or "≠", it has the minimum E = 12/5. If we are given the additional information that $a \leq b \leq c$, then Figure 2(v) is the most efficient solution.

Table 2 shows the number of boolean conditions like "$a = b$" or "$a \neq b$" evaluated for different triangle types for the pseudocode in Figure 2(iv). Here, the notation T(5) in column "$b = c$" means the condition "$b = c$" evaluates to T (true) in line 5 of Figure 2(iv). This gives E = (2+2+2+3+3)/5 = 12/5. A similar analysis of the pseudocodes in Figures 2(ii)-(iii) gives E = 16/5, which is 30% higher than 12/5. The pseudocode in Figure 2(i) has E = 35/5. The pseudocode in Figure 2(v) is a slight simplification of that in Figure (iv) and has E = 9/4; the only two cases of isosceles triangles are now: "$(a < c) \wedge (a = b)$" and

"$(a < c) \wedge (b = c)$". Since sorting a, b, c takes at least 2 comparisons, Figure 2(iv) is still the best and many students fail to obtain this solution.

Table 2. Computation of E = 12/5 for Figure 2(iv).

Triangle Type	Conditions $a = b$	$a = c$	$b = c$	Number of evaluations
Equilateral	T(1)	T(2)		2
Isosceles: case ($a = b$)	T(1)	F(2)		2
case ($a = c$)	F(1)	T(5)		2
case ($b = c$)	F(1)	F(5)	T(5)	3
Scalar	F(1)	F(5)	F(5)	3

```
1. if C1 then output("equilateral")
2. if C2 then output("scalar")
3. if C3 then output("isosceles")
```

(i) A very inefficient pseudocode.

```
1. if C1
2. then output("equilateral")
3. else if C3
4. then output("scalar")
5. else output("isosceles")
```

(ii) A better but not-so-good pseudocode.

```
1. if (a=b) AND (a=c)
2. then output("equilateral")
3. else if (a=b) OR (a=c) OR (b=c)
4. then output("isosceles")
5. else output("scalar")
```

(iii) An slightly better variant of (ii), with the nested then/else parts interchanged.

```
1. if (a=b)
2.    if (a=c)
3.    then output("equilateral")
4.    else output("isosceles")
5. else if (a=c) OR (b=c)
6. then output("isosceles")
7. else output("scalar")
```

(iv) A more efficient variation of (iii).

```
1. if (a=c) //assuming a <= b <= c
2. then output("equilateral")
3. else if (a=b) OR (b=c)
4. then output("isosceles")
5. else output("scalar")
```

(v) Another variation, assuming $a \le b \le c$.

Figure 2. Several alternate pseudocodes for the TRC-problem having different efficiency in terms of E.

III. CBSG-PROBLEM

Let $B(n, m)$ = the set of binary strings of length $n \ge 1$ and having m ones, $0 \le m \le n$. We want to generate the strings in $B(n, m)$ one by one in lexicographic order, say, to avoid generating a string more than once and missing some strings altogether. The lexicographic order $s < s'$ between two strings s, $s' \in B(n, m)$ means s represents a smaller integer than s'. For $B(4, 2)$, the lexicographic ordering is: 0011 < 0101 < 0110 < 1001 < 1010 < 1100 and thus we want to generate these strings in that order.

A. WBS for CBSG problem

Figure 3 shows a WBS for the CBSG problem, with work-units w_1 to w_4. The use of "leftmost" in w_2 is critical in two ways: (1) we sometimes cannot determine the "rightmost" position of change in s without determining the leftmost position of change, and (2) the corresponding new w_3 and w_4 do not give us any computational advantage. The leftmost position of change in s is the position of '0' in the rightmost "01" in s. If there is no "01" in s, then there is no string $s' > s$ and $s = 1^m 0^{n-m}$, the last string in $B(n, m)$. In this case, w_3 and w_4 will not be done. (This illustrates an important point about WBS: the child-nodes of a node do not always form an and-decomposition; they have to be, however, disjoint and one or more of them must always suffice to complete the parent-task. A program that implements a WBS must thus contain code for each child-subtask of a node because for some input situations the work for that child-subtask has to be done.)

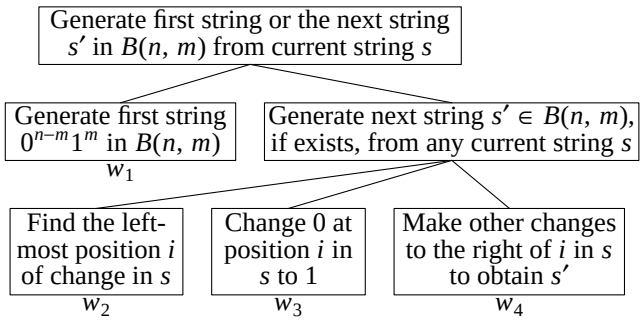


Figure 3. A WBS for CBSG problem.

B. From WBS to Pseudocode

To transform a WBS into a pseudocode with "stubs" for the work-units, one may need to add a good amount of control logic. We may also need to choose parameters of the functions for the main-task (root node of WBS) and some other tasks, and add some variables for use in the control-logic. Building a WBS first makes building a pseudocode easier. Figure 4 shows the pseudocode for the WBS in Figure ; "stubs" for the work-units are shown in bold. We use n and m instead of "length" and "numOnes" because of the small column-width of the printed text. The nextBinString-function returns NULL if there is no next binary string; otherwise, it returns a pointer to the binary string generated. The static-variable binString holds the most recent string (a beginning programmer may make binString a global variable instead).

 ISBN: 978-1-61208-230-1

```
char *nextBinString(int n, int m)
{ static char *binString = NULL;
  if (!binString) {
     allocate storage for binString;
     generateFirstBinString; }
  else {
     find leftmost position i of change in binString;
     if (i not found) binString = NULL;
     else { change 0 at position i to 1;
            make other changes in binString;
          }
  }
  return(binString); //NULL, if no next string
}
```

Figure 4. A pseudocode for WBS in Figure 3.

Figure 5 shows a pseudocode for generating all strings in $B(n, m)$. We can start a new cycle for strings in $B(n, m)$, perhaps with one or both of n and m changed, and call nextBinString(n, m) repeatedly once the previous cycle ends, indicated by the return-value NULL.

```
void genAllBinStrings()
{ char *binStr;
  int maxLength = 100;
  prompt("Enter string length n >= 1
          and <= maxLength");
  read(n);
  prompt("Enter number of 1's m >= 0
          and <= length");
  read(m);
  do { binStr = nextBinString(n, m);
       if (binStr) write(binStr);
  } while (binStr);
}
```

Figure 5. A pseudocode to generate all strings in $B(n, m)$.

We now present three ways of improving the efficiency of computing the next binary string s' from s.

C. Implementation/optimization issue: IS.1

A close look at the successive strings in $B(n, m)$ shows to get the next string s' from an s most of the time the changes to s are made to few of its rightmost bits. This means the search for the rightmost "01" in s should be done from the right to left. A naive code to find the position of '0' in rightmost "01" in s is shown below.

```
for (i=n-2; i>=0; i--)
    if (('0' == s[i])&&('1' == s[i+1]))
       break;
```

The problem with this code is that it looks at some of the trailing 0's in s more than once. For example, each of the three underlined zeros in s = 00111**0**1111000<u>0</u>0, where the desired '0' in s is shown in bold, are looked at twice by the above code. A better way to find the rightmost "01" in s is shown below; it looks at the same positions in s as before but they are now looked at only once. In both cases, if $i \geq 0$ on termination of the for-loop(s), then it gives the position of '0' of the rightmost "01" in s.

```
for (i=n-1; i>0; i--) //find rightmost '1'
    if ('1' == s[i]) break;
for (i=i-1; i>=0; i--) //skip preceding 1's
    if ('0' == s[i]) break
```

This code is an example of a general principle "Do A Little and Take Advantage of it" (in short, DALTA, to be read as "delta"). Another applications of this principle was in Figure 2(iv) in the construction of nested if-then-else. To force the better implementation of w_2 shown above, we decompose w_2 in Figure 3 as shown below.

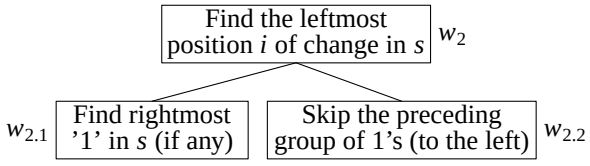


Figure 6. A decomposition of w_2 in Figure 3.

D. Implementation/optimization issue: IS.2

A further analysis of the relationship between a string $s \in B(n, m)$ and its next string s' shows that we can avoid the for-loop for skipping the ending group of 0's in s (if any) and, moreover, we need the for-loop to skip the preceding group of 1's only in one case. For this, we keep track of $z(s)$ = size of the ending group of 0's in s, $k(s)$ = size of the preceding group of 1's in s (which is the same as the rightmost group of 1's), and update them as follows:

Case $k(s) = 1$: $z(s') = z(s) + 1$, $k(s') \geq 1$.
Case $k(s) > 1$: $z(s') = 0$, $k(s') = k(s) - 1$.

E. Implementation/optimization issue: IS.3

Now, consider the work-unit w_4 in Figure 3. First, we change '1' of the rightmost "01" (at position $i + 1$) in s to '0'. Next, we move the remaining $(k(s) - 1)$ many 1's of the rightmost group of 1's to the extreme right in s and bring the ending groups of 0's in s (if any) to the left of these 1's. A naive approach for this would be to interchange left half of the last $n - i - 2 = k(s) + z(s) - 1$ positions in s with its right half. But a slightly more efficient method is to simply interchange $p = \min\{k(s) - 1, z(s)\}$ rightmost items in s with p leftmost items among the last $n - i - 2$ positions. Thus, for s = ···**01**<u>111</u>0000<u>000</u>, with $k(s) = 4$, $z(s) = 7$, and s' = ···**10**<u>000</u>0000<u>111</u>, we need to interchange only $p = 3$ positions (shown underlined in s) among the last $10 = 4 + 7 - 1$ positions, instead of interchanging $10/2 = 5$ positions. Since the rightside p items

 ISBN: 978-1-61208-230-1

will be made '1' and leftside p items will be made '0', we don't need to read these $2p$ items for the exchange.

F. Final code and its performance

Figure 7 shows a complete *C*-code for nextBinString-function based on Figure 4 and the optimizations in (IS.1)-(IS.3). This corresponds to version V.1 in Table 3; Table 3 also briefly describes three other versions. Table 4 shows some performance data from the actual measurements of average (over 5 runs) execution time for generating the strings in $B(n, m)$, without printing. Note that $|B(n, m)| = C(n, m)$ is largest when $m = n/2$. The version V.1 has the best performance. The left-to-right search for rightmost "01" in V.4 contributes most to its inefficiency. The recursive version V.3 has no search for "01" and the issues (IS.1)-(IS.3) are not relevant; it is less efficient than V.2 due to the cost of recursive calls. In V.3, we successively fill the positions $i = 0, 1, \cdots, (n-1)$ in the binary string in that order as follows using two recursive calls: fill position i by '0' (if $m < n$) and call nextBinString($n-1$, m) to fill the remaining positions on right, and fill position i by '1' (if $m > 0$) and call nextBinString($n-1$, $m-1$) to fill the remaining positions on right.

Table 3. Brief description of versions V1 to V4.

V.1:	Searches the rightmost "01" from right to left and adopts optimizations for (IS.1)-(IS.3); see Figure 7.
V.2:	Searches the rightmost "01" from right to left but does not adopt optimization for (IS.1)-(IS.3).
V.3:	Uses recursion to generate strings in $B(n, m)$.
V.4:	Searches the rightmost "01" from left to right and does not adopt optimization for (IS.1)-(IS.3).

Table 4. Aver. #(ticks over 5 runs in generating all strings in $B(n, m)$; a tick = 1/128 sec. The numbers in parentheses give aver. #(accesses to items of a string in $B(n, m)$) in search of the rightmost "01", if any.

Version	$n = 20$ $m = 10$	$n = 30$ $m = 10$	$n = 30$ $m = 15$	$n = 30$ $m = 20$
V1	0.6	83.6	419.6	93.6
	(1.0)	(1.0)	(1.0)	(1.0)
V2	0.8	156.0	739.8	164.0
	(4.2)	(5.4)	(4.3)	(4.4)
V3	0.8	168.2	783.6	172.2
	(3.8)	(4.3)	(3.9)	(4.3)
V4	3.8	895.4	4578.2	823.4
	(28.5)	(48.3)	(43.5)	(38.7)

IV. SOME ELEGANT CODING TECHNIQUES

We have seen in Sections II-III that going from an algorithm to an efficient implementation is a non-trivial task. We now briefly describe a few techniques to go a step further in creating a high quality elegant code, without sacrificing the efficiency, simplicity, and clarity. This involves some post-processing (cleaning) of the code, a step often ignored by students. One such technique is code-folding; we describe below two types of code-folding. See [1][4] for other techniques of good programming. In Section III.C, we have seen an example of the opposite process "code-unfolding", where we replaced a loop by two loops to reduce computation.

```
#include<stdio.h>

char *nextBinString(int length, int numOnes)
{ static char *binString;
  static int i, //posn of '0' in rightmost "01"
              //= -1 if there is no "01"
            k, //size(rightmost group of 1's)
            z, //size(group of 0's at length-1)
            firstCall=1;
  int j, min;
  if (1 == firstCall) {
     firstCall = 0;
     binString = (char *)malloc((length+1)*
                                sizeof(char));
     for (j=length-1-numOnes; j>=0; j--)
         binString[j] = '0';
     for (j=numOnes; j>0; j--)
         binString[length-j] = '1';
     i = length - 1 - numOnes; //= -1 if numOnes
                               //= length
     k = numOnes; z = 0;
  } else if ((-1 == i) || (0 == k)) {
           free(binString); binString = NULL;
           firstCall = 1;
  } else { //this part may set i = -1 or k = 0
           binString[i] = '1';
           binString[i+1] = '0';
           k--;
           if (k > 0) { //move 1's to right; set
                        //i, z for new binString
              if (z < k) min = z;
              else min = k;
              for (j=0; j<min; j++) {
                  binString[length-1-j] = '1';
                  binString[i+2+j] = '0';
              }
              i = length - 1 - k; z = 0;
           } else //no move of 1's needed; set
                  //i, k, z for new binString
                  { z = length - 1 - i;
                    for (i=i-1; i>=0; i--)
                        if ('0' == binString[i])
                           break;
                    k = length - 1 - i - z;
                  }
        }
  return(binString);
}
```

Figure 7. Final code for nextBinString-function.

A. Folding nested if-then-else

We often see student-codes that do not make proper use of else-statements. For example, the two if-statements "if (x > y) x = y; if (x <= y) y = x" can be simplified to "if (x > y) x = y; else y = x" to avoid the unnecessary test "x <= y". An extreme case of code-folding is replacing an if-then-else statement by a simple statement as in replacing "if (0 == x) y = x; else y = 2*x" by "y = 2*x". Figures 8(i)-(ii) show two slightly more complex cases of code-

folding, where the code on the right gives the simplified form. The best solution to the TRC-problem in Figure 2(iv) uses both folding and unfolding of if-conditions.

```
(i)  if (x)                 if (x && y)
        if (y) s = t;          s = t;
        else u = v;         else u = v;
     else u = v;

(ii) if (x)                 if (x && !y)
        if (y) u = v;          s = t;
        else s = t;         else u = v;
     else u = v;
```

Figure 8. Nested if-then-else simplified by code-folding.

B. Folding else-part

Given an array of non-zero numbers, suppose we want to compute posCount = #(positive items) and negCount = #(negative items). Figures 9(i)-(ii) show two inelegant student-codes for this problem. The first one ignores that the array-items are non-zero, and both of them ignore the property "posCount + negCount = #(items)".

```
(i)   for (posCount=i=0; i<n; i++)
          if (nums[i] > 0) posCount++;
      for (negCount=i=0; i<n; i++)
          if (nums[i] < 0) negCount++;

(ii)  for (posCount=i=0; i<n; i++)
          if (nums[i] > 0) posCount++;
          else negCount++;
```

Figure 9. Two inelegant solutions to a simple counting problem, where each nums[i] ≠ 0.

The second solution is more efficient than the first, but the best solution shown below is obtained by the DALTA-principle. It is missed even by some graduate students.

```
for (posCount=i=0; i<n; i++)
    if (nums[i] > 0) posCount++;
negCount = n - posCount;
```

C. Folding or merging of loops

Consider computing the variance V of nums[i], $0 \le i < n$. To use the formulas $V = [\sum_{i=0}^{n-1}(\text{nums}[i] - a)^2]/n$, where $a = (\sum_{i=0}^{n-1} \text{nums}[i])/n$, we need two separate loops. If we use the formula $V = (\sum_{i=0}^{n-1} \text{nums}[i]^2) - a^2$, then a novice programmer may also use the two loops shown below to compute $\sum_{i=0}^{n-1} \text{nums}[i]^2$ and a.

```
for (sum=i=0; i<n; i++)
    sum += nums[i];
for (sumOfSquares=i=0; i<n; i++)
    sumOfSquares += nums[i]*nums[i];
```

A better solution is to merge the two loops. This code is shorter and runs a little faster; it performs half as many tests "$i < n$" and half as many increments "i++".

```
for (sum=sumOfSquares=i=0; i<n; i++) {
    sum += nums[i];
    sumOfSquares += nums[i]*nums[i];
}
```

V. CONCLUSION AND FUTURE WORK

We have presented here a new approach to teaching undergraduate-level programming by using the notion of work-breakdown structure (WBS) as an intermediate step in applying the well-known stepwise-refinement method (SRM). Our four-step approach consists of: (1) creating a WBS of tasks in solving the problem, (2) adding control logic at various levels of the WBS to build a high-level pseudocode, (3) creating an efficient implementation of each work-unit (lowest level tasks in the WBS) based on a detailed analysis and its alternative implementation choices, and finally (4) applying certain code-transformations to obtain a more elegant program without loss of efficiency and logical clarity. Our initial experience in using this approach has been very positive. The students designed more efficient and elegant code using less time and with fewer errors. Our future work will involve a more extensive study of this new approach on a larger student population and extending the new approach to OO-programming.

REFERENCES

[1] J.L. Bentley, *Programming Pearls* (2nd ed.), Addison-Wesley, 2008.

[2] D. Gries, What have we not learned about teaching programming. *IEEE Computer,* Oct. 2006, pp. 81-82.

[3] P.C. Jorgensen, *Software Testing: a craftsman's approach* (3rd ed.), Auerbach Publ., 2008, pp. 22.

[4] A. Hunt and D. Thomas, *The Pragmatic Programmer,* Addison-Wesley, 2000.

[5] V. Preoteasa and R.-J. Back, Invariant diagrams with data refinement, *Formal Aspects of Computing, 24(2012), pp. 67-95.*

[6] R.-J. Back and J. von Wright, *Refinement Calculus: a systematic introduction,* Springer Verlag, 1998.

[7] C. Morgan, *Programming from specifications (2nd ed.),* Prentice-Hall, 1994.

 ISBN: 978-1-61208-230-1