A Study of Programming Skill Development in Education

Peraphon Sophatsathit
Department of Mathematics, Faculty of Science
Chulalongkorn University
Payathai Road, Bangkok 10330
Email: sperapho@chula.ac.th

Abstract: Classroom programming assignments have always been regarded as unreal, no commercial value, and to some extreme, nonsense work that cannot be taken into consideration for work experience. One of the prevalent rationale behind this problem is because the programs are written by unskilled, neophyte authors—the newly arrived college freshmen. The stigma of such discredited belief does not come without any proving ground. The fact that most students treat those programming assignments as the rest of "homework" to be finished and forgotten entails their ignorance about the craft of programming. This study investigates the fundamental building blocks of programming skills in academic environment that must be fostered in the repertoire of these young apprentices of computer science.

1. Background and Motivation

Many novice programmers, as well as some experienced but non-rigorous ones, practice quick-and-dirty programming approach for solving various problems. The approach is usually carried out by creating one piece of run-on code from scratch in front of a terminal. When editing is finished, a compile-and-go step commences merely to implement some algorithms at hand. Subsequent after-thought modifications are pursued by piling more code onto the existing one, with little consideration for maintenance. When it is all said and done, the code is often deleted or shelved which is eventually forgotten. In case where there is enough interest to pursue the work, the original code is virtually impossible to reuse. The recipient ends up starting everything afresh. In a typical classroom situation, especially for most introductory classes, the issue of reuse is out of the question. Every program being written is built from scratch in one terminal-sitting, compile-and-go style of programming simply to finish the assignment in time. Program planning never comes into play.

Despite the religious exercise of structured programming technique, this type of monolithic programming practice precludes a number of advantages precipitated from the application of systematic software engineering principles, e.g., software reuse, team programming, maintainability, comprehensibility, configuration management, and most important of all, good programming practice.

The importance of "good" programming practice is reflected by the programmer's skills. One example of such skills is the proper incorporation of routines into program development. Routines are perceived by most students and novice programmers as a means for dividing program into manageable chunks fitting roughly in one screen. There is little concern about the role of routines and why they must be used. McConnell [4] elaborates the enormous contributions of routine to programming and calls it "arguably the single greatest invention in computer science."

This study investigates some fundamental shortcomings and misconceptions about programming education, skill development, and good programming discipline in academic environment. Section 2 of this paper describes some well-known guidelines for the experiment. Section 3 elucidates the principal objectives to be achieved, as well as the results of the study. Some concluding remarks are presented in Section 4, followed by recommended future course of action to establish an effective education program for Computer Science students.

2. Approach

The study employed a group of students from second semester programming course who have just been exposed to minimal programming techniques (approximately 4-5 assignments) in their introductory course. None of them has yet learned "the tricks of the trade" to be proficient in programming. The process began by examining every student's source code to explore how it was written. Tips and techniques were arbitrarily injected during result feedback and error discussion sessions. One of the goals in doing so was to help the students attain better programming skills. Most students, unfortunately, neglect this “homework assignment” and get it done as soon

as possible, regardless of any continuation in successive assignments. As a consequence, programming was just a tedious typing lesson.

In order to foster standard programming discipline and avoid such negligence, the assignments were laid out by emphasizing:

1. structured approach focusing on modularity;
2. design planning to achieve high cohesion module having single entry and single exit; and
3. modern software engineering paradigms such as reuse, modifiability, and comprehensibility.

The overall procedures encompassed the following stepwise planning and refinement as follows:

1. Established fundamental programming disciplines such as basic software engineering methods and methodical program development;
2. Conducted learn-by-example process to stimulate learning;
3. Maintained uniform classroom environment for inexperienced apprentices to gradually build up their programming skills and knowledge;
4. Applied controlled experiment using problem solving type of assignments; and
5. Emphasized on coding style to stipulate:
    - variable naming: meaningful and appropriate length (with the exception of i, j, k);
    - organization: structured, routine use, and well-defined module;
    - module size: small (without separate compilation such as Make file);
    - simple I/O and environment: limit only to simple I/O statements, namely, printf, scanf, and getchar, running under Unix operating system;
    - indentation: readable and consistent style [6]; and
    - documentation: comments on routine and block headers, and wherever deemed appropriate.

The source code was visually inspected based on the above coding style stipulation to assess the students' programming skills. One might contend that there are myriad of more efficient tools and techniques available for program inspection and analysis, e.g., beautifier tool, control-flow graph tool, profiler, program slicing, etc. The truth remains that as immature nature of software engineering is today, nothing will replace the good old eyeball inspection. Moreover, numerous well-established inspection techniques [2] have been widely used by software practitioners. As such, the basic premise of this study relies on that fact that as coding style becomes more consistent, the program becomes more readable and comprehensible [6] which reflects the author's skill improvement.

3. Experimentation

Ten programming assignments were given to a group of first year Computer Science students in Programming Techniques class (using C language as a means). The assignments ranged from simple constructs such as basic I/O statements, selection (if, switch), repetition (while, for, and do-while), to complex constructs such as function invocation, coupling, aggregate structures, recursion, and file processing. Various software engineering principles were introduced to instill systematic programming discipline such as modularity, program design, and testing. Despite the absence of formal introduction of systematic test procedures (e.g., white-box testing, black-box testing, regression testing, and Verification, Validation, and Testing (VV&T) [3], all of which are beyond the scope of this course), a few fundamental testing aspects were established. Some of the predominant techniques are output format conformance, typical run-time errors (e.g., pre-matured end of file or no input data, wrong type, index out of range, etc.), and code reuse. The code reuse aspect, in particular, with increasing importance in software R&D [5], was incorporated through two consecutive assignments (2 & 3) where the succeeding assignment stressed feature enhancement of the preceding one. Consequently, the notion of code reuse was unscrupulously exercised.

A solution to every assignment was posted on-line for all students to learn proper programming style according to the aforementioned guidelines. Some sample solution code segments are illustrated below in Figure 1-5 based on the above guideline stipulations Figure 1 exemplifies appropriate variable naming scheme. Program organization such as defined constants, comment style, standard conventions, etc., is given in Figure 2. Figure 3 depicts the structure of a fundamental program unit, i.e., module. Sample I/O constructs are shown in Figure 4, emphasizing simple and straightforward I/O operations. Figure 5 illustrates one of the most important aspect of good programming practice, that is, program documentation.

```
int             in_val[Max];
int             val_index[Max];
float           mid, avg;
double          std;
```

Figure 1: Variable naming

```
#define         Max             70
#define         Assgn           10
#define         Len             11
#define         Succeed         1
#define         Failure         0

/*
 * student's record
 */
typedef         struct          rec
{
                char            ID[Len];
                int             hw[Assgn];
                int             mid;
                int             final;
                double          total;
} Student;

/*
 * prototypes
 */
int             init(Class *);
int             read_in(Student *);
void            sort(int, Student *);
void            swap(Student *, Student *);
void            compute(Student *, int);
void            class_stat(Student *, int, Class *);
void            cleanup(Class *);
```

Figure 2: Organization

```
/*
 * in  : input data array
 * out : the number of data actually read in
 * desc: this function reads input data from "stdin" using a local
 *       variable. The value is then put into the input data array.
 */

int
read_input(int *ival)
{
        int     val, cnt = 0;

        while (cnt < Max && scanf("%d", &val) != EOF)
        {
                ival[cnt++] = val;
        }
        return cnt;
}
```

Figure 3: Module size

```
while (scanf("%f%f", &first, &next) != EOF)
{
        printf("The sum of %.2f and %.2f = %.2f\n", first, next,
                first + next);
        printf("The difference of %.2f and %.2f = %.2f\n", first,
                next, first - next);
        printf("The product of %.2f and %.2f = %.2f\n", first,
                next, first * next);
}
```

Figure 4: Simple I/O

```
/*
 * Written by: Peraphon Sophatsathit
 * Date Written: Aug 13, 1998
 * Description: There are two main points to be demonstrated in this
 *              assignment, i.e., static data type and dynamic memory allocation.
 *              The size of the working array is dynamically determined by the
 *              size of the fixed static array.  As the number of elements in
 *              the static array increases/decreases, the corresponding size
 *              of dynamic array changes accordingly.  This is just "one"
 *              useful techniques for handling varying size arrays.
 */
```

Figure 5: Indentation and documentation

Categorical weights ranging from 0 to 3 were imposed to gauge the students' performance, where zero being undefined or not available (N/A), one being poor, two being fair, and three being good.

Assignment #1 was a type-and-go assignment designed to observe the students' initial untrained coding style. Assignment #2 & #3 were related so as to exercise code reuse and enhancement. Assignment #4, #5, and #6 covered typical procedural programming constructs. Routines were first introduced in assignment #7. Assignment #7 and #8 were somewhat involve which resulted in many incomplete submissions, together with numerous patches and sloppy code. Structure type and file processing were introduced in Assignment #9 and #10, respectively. The comparative coding results are summarized in Table 1. Table 2 depicts the raw scores collected per assignment.

Table 1: Comparative summary of coding stipulation

No.	Naming	Indentation	Organization	Module size	I/O	Documentation
1*	1	1	0	0	0	0
2	1	2	2	0	1	1
3	2	3	2	0	2	2
4	2	3	3	0	2	2
5	2	3	3	0	2	3
6	3	3	3	0	3	3
7	2	3	2	2	3	2
8	2	3	2	3	3	2
9	3	3	3	3	3	3
10	3	3	3	3	3	3

Table 2: Raw scores by Assignment

Assignment	1	2	3	4	5	6	7	8	9	10
Average	10.0	9.08	8.45	7.70	7.73	8.18	5.32	6.80	7.73	7.97
S.D.	0.00	1.70	2.59	3.09	2.81	3.73	4.41	4.09	4.14	3.87
Max	10	10	10	10	10	10	10	10	10	10
Min	10	5	4	1	2	0*	0*	6	0*	7

* obtained from the students who did not turn in the assignment.

Hindsight analysis from the students' source code and their responses revealed that the most difficult task encountered was debugging. As mentioned earlier, they were unaware of any systematic test procedures to conduct in-depth test coverage. Instead, they painstakingly debugged the code line by line. As a consequence, the majority of time spent was wasted on brute-force debugging activities—a misconception students and novice practitioners regard as part of test process that in fact is entirely different from systematic testing [1].

4. Conclusion

It was evidence from this study that the fundamental aspects of Software Engineering were relatively straightforward to convey. Students have no trouble grasping the principles and deposit the knowledge into their programming repertoire. The challenge of fostering good programming practice on the students' part is how to establish systematic program planning through analysis and design process. The biggest misconception in most people's mind is "everyone can be a programmer." The fact is that only the rigorously trained ones who methodically practice these software engineering principles will become "good" software engineers.

5. Future Work

Despite strong emphasis on structured programming, modular program design, and careful documentation, skill improvement depends primarily on individual's efforts. The stigma of classroom assignments will never be eliminated if the students are not motivated to improve their programming discipline. The question is how the skill can be systematically and effectively learned and applied. Another issue of skill development in academia can be addressed by the use of tools in classroom environment. The students should be exposed to helpful and effective development tools so that they can concentrate all their efforts on more important aspects of program development such as problem analysis, program design, and testing. Classroom assignments will soon be comparable to their real-world/commercial software development counterpart.

6. References

[1] Beizer, Boris. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1984.

[2] Gilb, Tom and Graham, Dorothy. *Software Inspection*. Addison-Wesley, 1993.

[3] Kitchenham, Barbara and Linkman, Steve. Validation, Verification, and Testing: Diversity Rules, *IEEE Software*, 46-49, 1998.

[4] McConnell, Steve. Why You Should Use Routines…Routinely, *IEEE Software*, 94-96, 1998.

[5] Mili, Hafedh, Mili, Fatma, and Mili, Ali. Reusing Software: Issues and Research Directions, *IEEE Transactions on Software Engineering*, 21(6): 528-558, June 1995.

[6] Sophatsathit, Peraphon. *Recommended C Style and Coding Standards*. National Electronics and Computer Technology Center, 1997.