Using methods of program engineering in the educational process

Shundeev Alexandr Sergeevich

Candidate of Physical and Mathematical Sciences
Lomonosov Moscow State University
(495) 939-21-59,
alex.shundeev@gmail.com

1 Introduction

Traditionally, an introductory programming course is the basis for studying computer science. Usually, the course is compulsory and includes both theoretical and practical training. Its basis is the study of a general purpose programming language, basic algorithms and data structures. In this paper, the programming language C (or C ++) is assumed. Further education in computer science may include a number of other special disciplines (operating systems, system programming, databases, network technologies, etc). All of them develop ideas and use the toolkit of the introductory programming course.

At present, the need to teach software engineering is practically not questioned. The main question is a way of introducing of this discipline into the educational process. Of course, a special educational course on software engineering is needed. It is necessary to decide at which stage of the education to include it in the curriculum. At the initial stage of education, some concepts of software engineering may not be understood. Students don't have practical programming experience and can't appreciate the importance of software engineering tasks and principles. The final stage of education may also contain risks. It should be noted that at the final stage of education practical classes in programming are either completely absent or have a very specialized form (numerical methods, statistical analysis of data). The lack of practical training can negatively affect the learning on software engineering.

Based on the above, the following approach may be of interest. The approach is to include suitable methods of software engineering in the introductory programming course. Subsequently, starting to study the special course on software engineering, the students will already have a certain amount of knowledge and skills. Undoubtedly, the goals and curriculum of the introductory programming course should not be changed.

Practical classes on programming reproduce the production model "customer" (teacher) - "performer" (student). The performer must develop and deliver a program to the customer in accordance with an assigned task. Consumed time is a key performance indicator of this production model. The software engineering principles and methods that reduce these time costs are reasonable to introduce into the educational process.

Let us illustrate this idea with the following example. Many teaching tasks have the following formulation. The input data are contained in the input.txt file. The student's

program processes the input data and writes the result to the output file output.txt. The process of debugging and delivering of the program includes the following sequence of actions that cyclically repeats:

- edit the *input.txt* file;
- program execution in a terminal;
- view the *output.txt* file.

Typically, the first and third actions are performed using some standard editor. The editor using (start the editor, open a file and close the editor) is a costly time-consuming operation. Note that in many cases, the use of the editor can be avoided.

To view the file, the following command can be used.

cat output.txt

This command is executed in the same terminal where the student's program is executed. To save input data (for example, the sequence of numbers 1 2 3 4 5) to the file, the following command can be used.

echo "1 2 3 4 5" > input.txt

Moreover, the considered sequence of actions can be represented by one compound command that is executed in the terminal.

echo "1 2 3 4 5" > input.txt; ./prog; cat output.txt

In this example, the knowledge (POSIX standard utilities, shell interpreter commands) that goes beyond the programming language is used. However, the use of these additional tools is justified. Their correct use significantly reduces the time of program development.

The example above shows us an acceptable approach that can be used for the introduction of software engineering principles into the introductory programming course. The approach is based on the use of a set of recipes that are designed in the form of self-contained learning examples. The article contains some of these examples that cover the following topics: debug printing, modular program structure and debugger.

2 Debug printing

Consider the following training task. It is required to write a program that takes three real numbers a (initial member of an arithmetic progression), d (common difference of successive members), e (precision) as inputs. The program [1] contains procedures for calculating the values of two parameters S_n^1 and S_n^2. These parameters represent the sum of the n first members of the arithmetic progression. The value of S_n^1 is calculated using the standard formula, and the value of S_n^2 is calculated by direct summation. It is required to find the number n for which the condition $|S_n^1 - S_n^2| > e$.

From the mathematical point of view, this task does not make sense. If we operate with real numbers, then the values S_n^1 and S_n^2 are equal. However, when a program is developed, real numbers are replaced by floating-point numbers. An arithmetic operation on floating-point numbers has a computational error. A computational error increases as the number of arithmetic operations is increased.

The solution of this task allows demonstrating all the main constructs of the C programming language. Also, it demonstrates the effectiveness of some software engineering methods and practices.

On some input data the program can be executed for a long time. For example, on the input data $a = 1.5$, $d = 10^{-3}$, $e = 10^{-7}$, the value $n = 30858$ will be found (Intel 64-bit processor, and the representation of real numbers in the form of double values). The need to control a progress of calculations leads to the need for debug printing. Instead of viewing an empty terminal screen, it is better to be able to analyze diagnostic messages that trace a computational process. For each successive number n, it makes sense to print the values n, S_n^1, S_n^2, $|S_n^1 - S_n^2|$ to the terminal screen.

Initially the output of the program was supposed to be the calculated value n that printed on the terminal screen. After adding debug printing, additional diagnostic messages are displayed on the terminal screen. It is required to separate the auxiliary debug printing from the text containing the output of the program. To do this, the standard output stream of error messages (*stderr*) can be used. The output of the program will be printed to the standard output (*stdout*). The diagnostic and error messages will be printed to the *stderr*. By default, both streams print their messages to the terminal screen. However, this behavior can be changed. These streams can print their messages to different devices. For example, when the program is executed by using the following command, the *stderr* stream messages will be saved to the *e.log* file.

./prog 2> e.log

The stdout stream messages will continue to be displayed on the terminal screen.

3 Modular program structure

Let us analyze the structure of the program [1] under consideration. Two relatively independent parts (two modules) can be distinguished. In the first part of the program, the possible values of the parameter n are sequentially examined, and the condition $|S_n^1 - S_n^2| > e$ is checked. The second part of the program contains the procedures for calculating the values of S_n^1 and S_n^2. These parts can be made in the form of separate files, which will subsequently be compiled separately. Thus, the program takes the modular structure. The separation into modules significantly reduces the build time of the resulting executable file. One doesn't need to recompile the entire program after making a local change. Only the module that was modified will be compiled.

The modular program structure provides the program code reusability. It becomes possible to avoid duplication of the program code. Let us change the condition of the original task. Now we will consider geometric progressions. The input data will be the

parameters b (initial member of a geometric progression), q (common ratio) and e (precision). The rest of the formulation of the task remains the same.

As a result, two programs are obtained. The structure of the new program is also logically divided into two modules. The new program can use the first module of the first program. It is only necessary to implement the second module that calculates the values of S_n^1 and S_n^2 in accordance with the definition of geometric progression.

4 Debugger

Previously, the advantages of using debug printing for logging diagnostic messages were shown. Debug printing can also be used to identify the causes of abnormal program behavior, including an abnormal program termination. However, in this case, the efficiency of debug printing noticeably decreases. Indeed, it is impossible to foresee the place in the program where the diagnostic message should be printed. It is also difficult to predict the list of parameters whose values should be printed. Usually, it is required to try many variants.

To solve this problem, the use of a debugger is more appropriate (for example gdb). It doesn't mean that one need to learn all the methods of working with the debugger. It is enough to collect a set of typical situations of abnormal program behavior. For each typical situation, one can put together a small set of tools that can lead to its resolution.

The most common of these situations is an abnormal program termination. If one runs the program from under the debugger then the program will be stopped at the point where the error occurred. The corresponding line number in the text of the program will be printed. To perform these actions, one only needs to know one debugger command (run). Hereafter, to eliminate the abnormal situation, one can again use debug printing. At the same time, after having studied one more debugger command (print), one can check the correctness of the values of local variables without using debug printing.

The described solutions can be demonstrated by the example of program with geometric progression. On the input data $b = 1.5$, $q = 2.1$, $e = 10^{-7}$ the program works correctly. The slight modification of the second parameter ($b = 1.5$, $q = 2.$, $e = 10^{-7}$) causes an overflow floating point exception. The input parameters $b = 1.5$, $q = 10^{-3}$, $e = 10^{-7}$ cause an underflow floating point exception.

5 Conclusion

The author has been conducting the practical classes on the introductory programming course at the faculty of mechanics and mathematics of Moscow State University. The engaging of software engineering principles makes it possible to increase the effectiveness of practical classes. Nevertheless the engaging of software engineering principles is a difficult organizational and methodical problem. The article describes the author's approach to solving this problem.

References

1. The source programming code of the training task. - https://github.com/group112/apsse2017