An Exploration into Effective Teaching of Introductory Programming

P. V. R. Murthy, A. Prabhakar
Faculty of Engineering and Technology, M. S. Ramaiah University of Applied Sciences, Bangalore 560 054
Contact Author e-mail: pvmurthy.cs.et@msruas.ac.in

Abstract

New Software based systems are being built in all engineering domains and their number is on the rise. Thus there exists a demand for high-quality professionals who can develop reliable and maintainable software for building such systems. It is also widely known now that, more often than not, software components are the weakest link in terms of being the triggers for system failures. In this context, programming is an activity that needs to be carried out with rigour and discipline to develop reliable and maintainable software. This article argues that it is only right learning of programming by our undergraduate students in our colleges and universities in India that can address the requirement of high-quality professionals being involved in developing software for various systems. However, this is possible only if right teaching methodologies of programming are practised by the teaching community in various science and engineering colleges in India. A survey of the foundational principles of procedural programming is carried out first and then teachers are given novel recommendations about how to collect data about the learning deficiencies of students in a systematic manner. Availability of such data can throw light on the wrong assumptions or stumbling blocks that act as an impediment in the effective learning of programming by students. Novel methods of carrying out self-review, by students, of their own programs and reviews of student programs by teachers during evaluation to identify classes of program logic errors that students usually commit are proposed.

Key Words: Programming, Test Driven Development, Mutation Operators, Structured Programming, Loop Invariant

1. INTRODUCTION

Software based systems are in wide-spread use today and there will be a much greater deployment of them in the coming decades. While there are several factors that play a role in the reliability of such systems, one critical factor is the reliability of software components. However, today, a significant percentage of programmers, while quite familiar with tools and technologies, are not well equipped with a sound foundation in programming that is good enough to build reliable and maintainable software components. The question arises what exactly are the causes of this situation. While part of the reason may be uninformed decisions and poor planning, a major cause is that software developers have not quite learnt programming the right way which certainly reflects on how programming is taught not only at various professional training centres but also at various educational institutions. Poor programming education, at least, partially leads to “software crisis” often encountered with frequently missed deadlines, cost overruns and buggy software.

First, we identify, in Section 2, the learning outcomes for a student of an introductory programming subject. Then in the subsequent sections, we delve into how a teacher can enable the students to meet the learning outcomes. A survey and compilation of effective ways of approaching programming that are recommended by expert computer scientists over decades leading to the development of reliable and maintainable software is carried out. Next, we propose that Test-driven development (TDD) [1] be introduced in introductory programming subjects. Most importantly, we propose that a novel method of self-review of programs based on a check-list be taught to students in introductory programming subjects. The check-list is derived from the mutation operator list [2] of a given programming language. Students can carry out self-review of programs as they are being written and they can also use it in the review of the programs of fellow students, as done in pair programming. In addition, we suggest that teachers may employ a review of the programs developed by students in class, assignments and laboratory exercises as well, so as to obtain insights into the gaps in learning the concepts of programming. An objective feedback of the difficulties in learning, by students, aids in improving or adjusting teaching methods dynamically during a semester or in future deliveries of the subject. We believe that effective and right methods of learning how to program machines is important for undergraduate students in all disciplines, for it is anticipated that with technological advances, today’s students need to carry out programming tasks in much greater numbers in future than the professional programmers of today.

While there are several aspects of programming that an article of this nature could deal with such as different paradigms of programming (procedural, functional, logic, object-oriented and feature-oriented) and their deployment on modern powerful processors, we deliberately limit ourselves to essentially procedural programming because this is what is taught to undergraduate students as per the curriculum in most Indian Universities. Hence, our focus is on the discussion of effective learning of the principles of structured programming which is applicable to programming languages such as C, despite its idiosyncracies, as it is still popular and used in undergraduate programming subjects. However, the emphasis is on developing an appreciation in the student’s mind about principles of programming, not restricted to a particular programming language. In addition to arriving at what needs to be taught based on a compilation of recommended ways of programming based on the foundational principles that evolved...
through pioneers in the field over decades, our suggestions for teaching programming effectively are

- Test Driven Development (TDD) be employed in teaching programming at least in a couple of lectures
- A logical error check-list, derived from the mutation operators of a programming language, be used by students as well as instructors while reviewing programs for defect prevention.
- Systematic collection of data about errors made by students can be used as an objective feedback to improve or adjust teaching of programming concepts. For students, it is an aid in self-review and peer review of programs.
- Additional emphasis be laid to make students understand the notion of the state of computation, fairly early during the semester
- Regular customization of teaching methods by identifying, periodically, during each semester the wrong notions, or invalid assumptions, or stumbling blocks that students have in their minds that become impediments in the effective learning of programming.

2. INTENDED LEARNING OUTCOMES

We strongly recommend the list below as the intended learning outcomes for an introductory programming subject with the goals of preparing the students to be able to develop reliable and maintainable programs.

- Gain insights into the process of programming by appreciating the essential meaning of Programs = Algorithms + Data Structures [3]
- Apply the principle of step-wise refinement in designing algorithms and constructing programs
- Apply the principle of modularization in the construction of programs
- Build appropriate abstractions while solving a problem to the right level and not get lost in low-level implementation details to start with right away
- Map to relevant constructs of a programming language based on the steps in the algorithm or the data structures used
- Gain insights into how to use iterative constructs and recursion clearly appreciating the differences between the two
- Gain insights into the notion of state of computation and develop the analytical techniques(informally) to reason about programs, as they are being developed or modified
- Develop programs that are reliable, maintainable and efficient
- Practise the discipline of defect prevention
- Validate a program by running it on adequate tests derived from the specifications
- Debug failing programs using the notion of state of computation
- Work in collaboration with other programmers to create larger programs.

3. PROGRAMS = ALGORITHMS + DATA STRUCTURES

A program in a high-level language is a sequence of statements or instructions which when compiled and executed may produce output(s) on accepting input(s). Programs even measure up to millions of lines of code for applications today. What really is a program, in addition to the trivial definition given above, in order to obtain better insights? Programs = Algorithms + Data Structures. However, with demanding schedules as well as lack of awareness about methodologies of constructing quality programs, many practitioners do not pay attention to the significance of the above notion of programs in the guidance offered in constructing programs of high quality. A consequence is that there is no systematic attempt to design algorithms in alignment with appropriate data structures weighing the pros and cons of considerations such as efficiency of programs. Furthermore, usually, no conscious attempt is made to demonstrate, in an introductory programming class that the choice of data structures for a given problem is crucial in that of the algorithm and vice versa. We strongly correlate this gap in practice of programming by professionals as a consequence of the teaching and learning methods employed in academic institutions and training centres. The alignment between an algorithm chosen and the access pattern of the data structure is crucial for the efficiency of the algorithm.

4. STEP-WISE REFINEMENT

The step-wise refinement method and strategies such as divide and conquer for designing algorithms, for example, are some of the conceptual tools that must be applied by every programmer during program construction. Studies indicate that in practice there exist very long methods or functions in real code, due to a big bang approach in coding, making it difficult to maintain them. In step-wise refinement (using a top-down approach), requirement specifications of software under construction are mapped to top-level actions(functions) initially to start with and each action is refined further and this process is repeated until the lowest level actions are designed and implemented. The hierarchical decomposition cannot be partitioning of functionality just for the sake of it. The partitions need to make sense for the problem. According to Niklaus Wirth [3], the creative activity of programming, to be distinguished from coding, is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures.

5. ON BUILDING MODULAR SOFTWARE TO TAME COMPLEXITY

Parnas’s principle of modularity and information hiding needs to be taught firstly at the conceptual level and secondly at the language level of how modules and interfaces can be expressed in a language such as C.

6. ITERATIVE CONTROL STRUCTURES

Iteration is an algorithm design concept. The mundane aspects of a loop construct, that a student ought to learn, are setting up the lower and upper bounds for the control variable along with the increment step. In addition, a student must learn to write the body of a loop as per the specification. The more crucial aspect of a loop that a student benefits from learning is the concept of a loop invariant. The loop invariant may typically be expressed as a logical expression derived
from the immutable properties of the data structure or variables accessed and modified in the body of the loop.

7. **INСУLСАТІNG THE DISCIPLINE OF ZERO-DEФЕCT PROGRAMMING**

The NATO Conference on Software Engineering and the people who attended it inspired a great deal of research in the late 1960s and 1970s. The research contributions that followed are an axiomatic basis for computer programming languages and a theory of weakest preconditions which influenced formal reasoning of programs for correctness. Step-wise refinement, structured programming, design of the programming language Pascal for pedagogical purposes, guarded commands and communicating sequential processes are the other research contributions. The above concepts strongly influenced programming language design, programming methodology and program correctness approaches. David Gries [4] raises a concern that to a large extent, we have failed in incorporating those foundational influences into teaching programming and computer science. He also lamented that we do not discuss about language shaping mind and thought and the role of notation. The way the subject is taught in most educational institutions does not impart training on the programming process. As Gries points out, most introductory texts are program texts and not programming texts. Gries also says that we do not emphasize the simplicity and beauty aspects in the creation of programs. Research contributions in terms of the idea that a program and its proof should be developed simultaneously are not taught even in an informal way. Gries goes on to say that most graduating computer science students are unable to explain what a loop invariant is. There is a strong need to motivate students to have the ideal of zero-defect programming, however hard it might be to achieve it in practice, and equip them with the necessary programming principles. Gries also criticizes introducing programming to novices using a language such as C which is known for its idiosyncrasies, as it is a weakly-typed language.

There is a need for much greater emphasis on students inculcating the discipline of stating the purpose of variables ahead of their usage and using self-revealing names for the context of the problem. This is essential for readability of code which affects maintainability.

8. **LEARNING PROGRAMMING USING TEST DRIVEN DEVELOPMENT**

Which one of the following methods needs to be adopted by a teacher? Developing the whole algorithm or program such that it works for all possible inputs or scenarios right away, or, developing the algorithm or program incrementally test case by test case as in TDD (Test Driven Development)? Usually, it is the case that insights into designing an algorithm arise from considering specific cases and then generalizing the algorithmic steps to work for all inputs or cases. On the other hand, using the TDD approach, a programmer develops a program incrementally, test case by test case. The refactoring or code restructuring steps in TDD may eventually lead to the same algorithm structure as in the former approach but it is not yet established by the research community. However, for pedagogical purposes, both the approaches with and without TDD may be taught so that the student clearly sees the difference.

We first briefly illustrate how a student may be taught test-driven development using Fibonacci program [1]. In TDD, a test is designed first and the corresponding implementation of Fibonacci is developed to check whether it passes the test. The Fibonacci program is run to validate the result and the process continues by designing the next test and further incremental development of the algorithm/code for the new test and so on.

```java
/*Test case 1*/
public void testFibonacci() {
    assertEquals(0, fib(0));
    /*Fibonacci function for test case 1:*/
    int fib(int n) {
        return 0;
    }
    /*Test cases 1 and 2:*/
    public void testFibonacci() {
        assertEquals(0, fib(0));
        assertEquals(1, fib(1));
    }
    /*Implementation corresponding to test cases 1 and 2:*/
    int fib(int n) {
        if (n == 0) return 0;
        return 1;
    }
    /*Test cases 1-3:*/
    public void testFibonacci() {
        int cases[][] = {{0,0},{1,1},{2,1}};
        for (int i=0; i < cases.length; i++)
            assertEquals(cases[i][0], fib(cases[i][1]));
        /*Implementations of tests cases 1-3:*/
        int fib(int n) {
            if (n == 0) return 0;
            if (n <= 2) return 1;
            return 2;
        }
        /*Let us generalize the code (we do not show all the steps of TDD here).*/
        int fib(int n) {
            if (n==0) return 0;
            if (n==1) return 1;
            return fib(n-1) + fib(n-2);
        }
    }
```

9. **USING A LIST OF MUTATION OPERATORS FOR REVIEW OF PROGRAMS**

Mutation testing firstly deals with mutating statements/operators in source code of the software module under test, with the mutations drawn from a set of mutation operators for the language in which the code is expressed. Thus a set of mutated program versions are created. The main idea of mutation testing is to strengthen the test suite incrementally until the test suite kills all mutants (or all mutated version runs yield bugs due to the mutations).

Our main observation is that the set of mutation operators designed for each programming language such as C and Java reflect common programming errors related to logic such as

- Missing statements in algorithm or code
- Wrong variable used on the left hand side in an assignment
- Wrong relational operator used in a condition
- Missing initialization statement for a variable
- Wrong logical operator in a boolean expression
- While loop replacement by do-while loop
and so on. Our proposal is that students be taught how to review their own programs, against specifications, as they develop them. Specifically, we propose the following, in addition to the usage of the existing and relevant methods to review code by students at their desks as programs are still being developed. Our new recommendation is that the list of mutation operators for a given programming language be used as a checklist in the process of self-review of one’s own program to detect logical errors, even before programs are run by them. This is expected to go a long way in a student’s ability to prevent logical errors before programs are run thereby inculcating the discipline of forethought avoiding unnecessary debugging effort. This also inculcates the best practice of browsing through one’s own program critically with the intention of detecting logical errors as the program is being developed. Furthermore, research in automatically detecting logical errors is still being carried out wherein formal methods have a role to play. The suggested review has the potential of revealing unintended errors as well as errors due to wrong mapping onto a language’s statements/operators while expressing a chosen algorithm to the language during coding. Logic errors in programs can also be detected to an extent by the proposed review. The review may also be used in peer programming and review of others’ programs by students.

10. INSTRUCTOR’S ROLE

A teacher of an introductory programming subject has a major role to play in effectively imparting not only knowledge of all of the specific principles or concepts covered in the article so far and more but also in their application, both in the class and the laboratory. A teacher needs to take an active role in identifying the wrong notions about programming that students usually have, either because they learnt it already in the wrong way or some concepts in the subject look rather odd as per the knowledge they already possess from other subjects such as mathematics. For example, students often have a difficulty in appreciating an assignment statement of the form variable-name = expression. This may be primarily due to inadequate attention paid to the teaching of the load and store concept of the underlying Von Neumann machines or the stored program computers that we use today.

11. CONCLUSIONS

We have considered the important problem of effective learning of introductory programming by students. To achieve this, we mainly pointed out that some colleges or training centres may tend to teach programming as mostly a coding activity. If such an approach is used, students may not think it to be important to first dwell on the design of the algorithm behind a program and then code the algorithm into a particular programming language. Furthermore, we strongly believe that even in an introductory programming subject, the significance of the statement Programs = Algorithms + Data Structures has to be comprehended by students. The present article is confined to the consideration of teaching of procedural programming effectively based on the principles of structured programming. The principles of step-wise refinement, modularity and keeping programs as simple as possible, if effectively taught to students, go a long way in them developing professional capabilities to deliver reliable and maintainable programs in their careers. A research direction that we point out is about how to investigate, in a systematic manner, the stumbling blocks or wrong assumptions that students have while learning programming. While a teacher learns informally by experience through interactions with students what their difficulties are in learning and the application of the subject, to be really able to help students circumvent the stumbling blocks in learning programming, research effort in education is required along the lines indicated below. What is required is systematic collection of data pertaining to the wrong assumptions/common logical errors that students make which act as impediments in effective learning and application of their programming knowledge. Our research fills in this gap by recommending the usage of check-lists derived from mutation operators of a particular programming language in self-review as well as peer reviews of programs. Our initial investigation based on the proposed approach reveals that students make logical errors related to (i) missing initialization statements, (ii) using a wrong or unintended variable in an expression, (iii) missing statements that affect the logic of a program and (iv) not realizing inefficiency introduced into a program by computing the same unchanging expressions in each and every iteration of a loop. The above efficiency related error can be detected by an intelligent compiler and may be considered related to mutation operators from the point of view of efficiency, a non-functional requirement. Our future research direction shall be in developing methods and templates to systematically collect data, during each semester on a continual basis, about the gaps inferred from reviews of programs written by students in an objective manner. Our research shall also focus on mapping the data about the gaps observed onto concepts/principles of programming and devising better ways of teaching those making learning outcomes actually realizable by students. Furthermore, our work shall continue to be concerned with how to help students acquire programming skills not influenced by a particular programming language and their thinking and creativity not constrained by a paradigm of programming, such as procedural programming.

REFERENCES