Teaching Novice Programmers Programming Wisdom

Dr. Randy M. Kaplan

Kutztown University of Pennsylvania
Kutztown, PA 19530
kaplan@kutztown.edu

Keywords: POP-I.A. learning to program; POP-II.A. novices

Abstract

Teaching students how to write computer programs has remained a challenge. Whether a student is new to programming or has some experience they often have not had enough to develop strategies for solving problems in a computer programming setting. It is akin to being in a rowboat without any oars. Yes, you can get there; it will just take a long time.

There has been a great deal of research over the years into the psychological and cognitive aspects of programming. The reality is that much of this research has not informed our teaching and we are still teaching programming as we did many years ago. Although the academic community may take offense at this statement looking at the way programming is taught today and applying the relevant research to the process can justify it. Without going through that process, it remains easy to see. Programming is largely taught today the way it was taught 60 years ago.

Although this paper is entitled "Teaching Novice Programmers Programming Wisdom," the wisdom referred to are encapsulations of some of the meta-skills that are needed to successfully write computer programs. The list is not meant to be in anyway comprehensive, complete, or all encompassing. It is a list used with some success in entry level programming courses. The purpose of writing and presenting the idea of wisdom for programming is to allow the computer science/education/psychology communities to comment on this wisdom and possibly offer other "nuggets" and approaches that might be employed during novice programming instruction.

1. Introduction: The Need for Wisdom

There are two main issues that are key to successfully teaching students to write computer programs. The first is how do you convey the information needed by the student actually write programs. This is akin to learning how to speak a new language; learning the words of the language and how to construct sentences in the language. The other issue concerns the strategies that students can use while they are learning and using the new language. These so-called meta-strategies go a long way to make learning and using a new language an easier process.

There have been many studies of programming (Blackwell 2002), how people learn to program (Buckingham, Hynd et al. 2004), how novices learn to program (Mayer 1981), and the psychological and cognitive aspects of programming (Sheil 1987). There have also been studies of programming languages that attempt to identify characteristics that may make them easier to learn (McIver and Conway 1996).

Those of us that teach programming have a tremendous challenge. We are required to teach a skill that requires us to impose rules for problem solving in a foreign language (a programming language). We teach programming much in the way we taught it from when programming was first taught. We tell the student about the tools and then we explain how to use the tools to solve specific problems. We do not address the necessary overarching or meta-skills that are required in the programming process. David Gries (1974) states that we do not explicitly teach problem solving when we teach people how to

program. Gries observations and the context in which he gives them are important. The comments were given some 36 years ago without supporting evidence or research.He goes on to say "the bright students somehow catch on …" Polya's work (1988) about problem solving stands by itself as a concise model of the kinds of heuristics that are extremely useful to students learning to solve mathematical problems. There is nothing special about the heuristics that Polya provides. The uniqueness of his book is that he sets down on paper certain principles for the teacher and for the student that are extremely useful for both. When reading Polya's book one is continually reminded of their familiarity, yet somehow having them documented in this way makes them more tangible and useful.

In this paper, following in the footsteps of Gries and Polya, statements will be made based on our experience with programming and teaching novice students to program.

When teaching a CS1 course it becomes apparent that most students do not have or are not aware of the meta-cognitive skills that are needed for programming. It became clear that an additional "pre-CS1" course would help make the CS1 course more accessible to students taking it. The purpose for CS0 was to level the playing field for all students. This course afforded an opportunity to cover some of the aspects of programming that are missed in the typical CS1 course. Specifically we would have an opportunity to incorporate teaching of the meta-skills necessary to carry out the programming task. The dilemma was how to present these meta-skills in such a way that they can be remembered and used by the students for the programming process. This paper describes the formulation used for presenting these meta-skills.

2. Introduction: Codifying Wisdom

It would be convenient if we could simply impart wisdom by codifying it in some way that (a) everyone remembers, and (b) everyone uses. In this sound byte, 10 second news story, limited attention span world, it would in fact be excellent because then, when we teach programming, we could impart this wisdom so our students would not find programming as frustrating as they do. Unfortunately codifying the experience of programming is not an easy thing to do.

There are many attempts at codifying programming wisdom. Through the years there are some notable artifacts of programmer wisdom. Examples include Frederick Brooks classic, "The Mythical Man-Month (Brooks 1995)," "201 Principles of Software Development (Davis 1995)," and "Joel on Software (Spolsky 2004)" are all examples of attempts to encapsulate some lessons learned (wisdom) for the purpose of improving the software development process.

Each of these venerable works contains important wisdom for programmers and system developers. On average each has 300 pages and would require a course to cover their contents. For the novice, much of their contents would probably be irrelevant to their task. Another work, "Code Complete" (McConnell 1993) is an excellent text about how to make sure that the code that a programmer writes works. This work (and subsequent editions) are also must reads for the programmer although not very appropriate for the novice programmer.

Statements of wisdom, also known as heuristics, can be gleaned from experience and if properly expressed can be immensely useful to the student. It is often said that the problem with novice students is not that they need meta-strategies; it is more that they need basic domain knowledge. We would claim that this is not an either/or situation but actually an "and" situation in that novices need both kinds of knowledge – especially when it comes to computer programming.

The heuristics or wisdom described in this paper bears some relationship to the work on meta-cognition in programming. Meta-cognitive skills are those that are used as strategies for solving problems or carrying out tasks. These may seem to be intuitively useful in the case where the situation is unfamiliar to the student or practitioner but this is not a confirmed fact. Research carried out by Shaft (1995) shows that when experienced programmers used meta-cognitive skills when attempting to carry out programming understanding, the use of these strategies did not improve the understanding of the programs.

There has also been research investigating what meta-skills may be used when programming. Uncovering what these skills might be has been somewhat allusive. One step in this process would be to classify the level or kind of skill needed to perform a specific programming task. Shuhidian (Shuhidan, Hamilton et al. 2009) attempted to do this but found that classification was more difficult than expected.

3. This Paper's Contribution

How can we codify the meta-skills of programming in a form that can be easily remembered and used by novice students? This was the challenge for a CS0 course. This paper defines a codification of programmer's wisdom suitable for the student beginning their computer science or information technology programs. In fact, this codification is one that can be continuously used by the student. Students who have graduated from our programs have commented that they continue to use these meta-skills. The contribution of this paper is the definition of the codification of a programmer's heuristics.

4. Programmer's Wisdom

4.1 Precedents for Experiential Wisdom (Heuristics)

One of the most notable examples of a presentation of experiential wisdom is the work of Polya mentioned earlier. Early in his work Polya specifies four phases for finding a solution to a problem. Comparable to some aspects of the programming wisdom to follow, Polya identifies the phases as understanding the problem, planning for solving the problem, execution of the plan for solving the problem, and lastly a "looking back" at the process used to solve the problem (Polya 1988), What justifies these phases? Who are they attributed to? Surely they existed before Polya. Polya next explains each of the phases in greater detail. For example, one of the statements made, "The students should consider the principle parts of the problem attentively, repeatedly, and from various sides(Polya 1988)." Again the question arises, where did this come from?

When reading Polya it is clear that the source of these heuristics was his own experience over. The present work has similarities to Polya's. It is an attempt to codify heuristics and wisdom that can be used by novice programmers to solve programming problems much as students solving mathematics problems can use Polya's work.

4.2 Wisdom Explained

In this section the "nuggets" of wisdom (heuristics) will be introduced and explained. Before introducing them, some comments are in order about the kinds of skills that these statements address. We can identify two kinds of statements. The first type of statement is one that is relevant to the programming process. The second type of statement is of a more general type that would apply to both writing computer programs and problem solving. Rather than consider this distinction a strict dichotomy between domain-relevant statements and more general problem-solving statements (or meta-cognitive statements) we can consider a continuum between these two classifications. A particular statement is to a lesser or greater extent of one or the other classification. We assume that statements having more to do with the programming domain will be on the one side of the continuum while those having more to do with meta-cognitive skills on the other side of the scale. Figure 1 shows a classification of the statements of programming wisdom on the continuum.

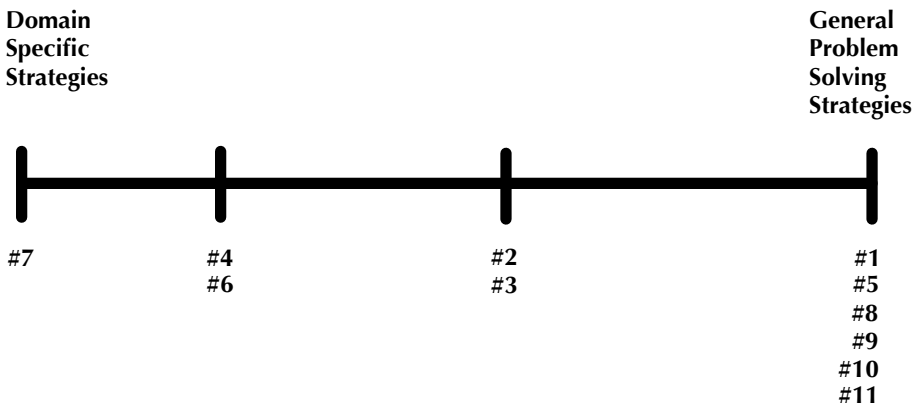


Figure 1 – Continuum of Programming Strategies vs. General Problem Solving Strategies

The codification of the programmer's wisdom is divided into 11 statements. There is nothing significant about this number except that it had to be a small enough number of statements to remember. Students readily memorize the 11 statements without difficulty. The 11 statements are as follows.

1. BREAK IT DOWN
2. LEARN THE BASICS WELL, LOOKUP THE REST
3. FIND AND USE EXAMPLES
4. TEST TEST TEST (and test some more)
5. FOLLOW DIRECTIONS
6. SOLVE THE PROBLEM IN ENGLISH FIRST
7. REPRESENT WHAT?
8. WHAT DOES IT DO?
9. MAKE SURE YOU UNDERSTAND THE PROBLEM
10. IF YOU CAN'T SOLVE THE BIG PROBLEM SOLVE A SMALLER ONE.
11. HAVE A PLAN B

5. Wisdom Defined

The 11 statements listed above are explained below. When the students are introduced to the 11 statements, they are given an explanation of each statement. The descriptions below are in lieu of the actual classroom explanations.

Break It Down

When you are faced with a problem that is too complex, one way to approach it is to break the problem down into smaller parts. By breaking the problem into smaller parts you may be able to more readily solve the original problem by solving its pieces.

Learn the Basics Well, Lookup the Rest

Beginning programmers often get hung up on how to do certain things. Because programs have to be very precise it is easy to make mistakes when you first start out. The point is that it is okay to look up what is needed when a student becomes stuck.

Find and use examples

There are thousands upon thousands of coding samples that can be examined to better understand how a specific task is accomplished. Using examples to create programs does two things; first you get to see how something is done, and if you pay attention by seeing how it is done you will learn how to do it. Second, you'll spend less time, at least at the start, writing your programs. Students complain about how long it takes to write a program. If you can find and use examples and you can learn from the examples you might be able to learn faster then if you tried to write a program from scratch.

TEST TEST TEST (and test some more)

Testing is part of programming. You test to get a program to work and you test to make sure the program works correctly. The better you test, the better your program will be.

Follow Directions

Make sure that you are doing what the directions say. The directions tell you what you can and cannot do. Don't overcomplicate or oversimplify what you have to do. Sometimes the directions will even give you part of the solution – so read carefully.

Solve the problem in English (or your native language) first

Learning a new programming language is similar to learning a new foreign language. You just don't jump right into writing the steps in the new language. That is a sure fire recipe for failure. Solve the problem in your native language. Write each of the steps in English or the language that you are most comfortable with. Translate this version into the programming language you are using. That way you will have the problem solved and you can focus on how to express your solution in the programming language.

Represent What?

An important part of solving any programming problem is to figure out how the problem will be represented in data. The variables that are needed and other structures like arrays need to be defined. This is a key aspect of creating a program to solve any programming problem.

What does it do?

Every computer program that was ever written does something. That is the nature of programs. They make computers do things that are desired by the programmer. Now sometimes the author of a program will get it wrong. The computer will do something that was not wanted. There can be lots of reasons for this and one of them is that the programmer really didn't understand what the computer was supposed to do for solving the problem. Make sure you understand what it is that you want the computer to do.

Make sure you understand the problem

If you don't understand what the problem is, how can you come up with a solution? You can't. Besides following directions, understanding what it is you must do is extremely important. If you don't understand what to do, don't even think about starting to work on the solution to the problem. Do whatever it takes to understand the problem you must solve.

If you can't solve the big problem, solve a smaller problem.

This may seem like the first statement (Break It Down) but it isn't. This has to do with complexity of the problem you are to solve. A problem might be so complex that you have no idea how to approach it let alone solve it. So, instead of bashing one's head against a wall (a tried and true method for solving problems), formulate a simpler problem related to the original problem. The simpler problem will be

solvable (we hope). Use the learning that came from solving the simpler problem to solve the more complex problem.

Have a Plan B

No matter how well we plan and no matter how completely we plan, there is always something that happens to get in the way of our plan. Therefore if you have a way to solve the problem and you find you have travelled down a path that doesn't have a good or desired ending, you need to have an alternative. Plan B is that alternative. It may be an alternate way of solving the problem. It may be a different breakdown of the problem. It may be a simpler approach. Whatever it is, have a plan B just in case.

6. Why These 11 Statements?

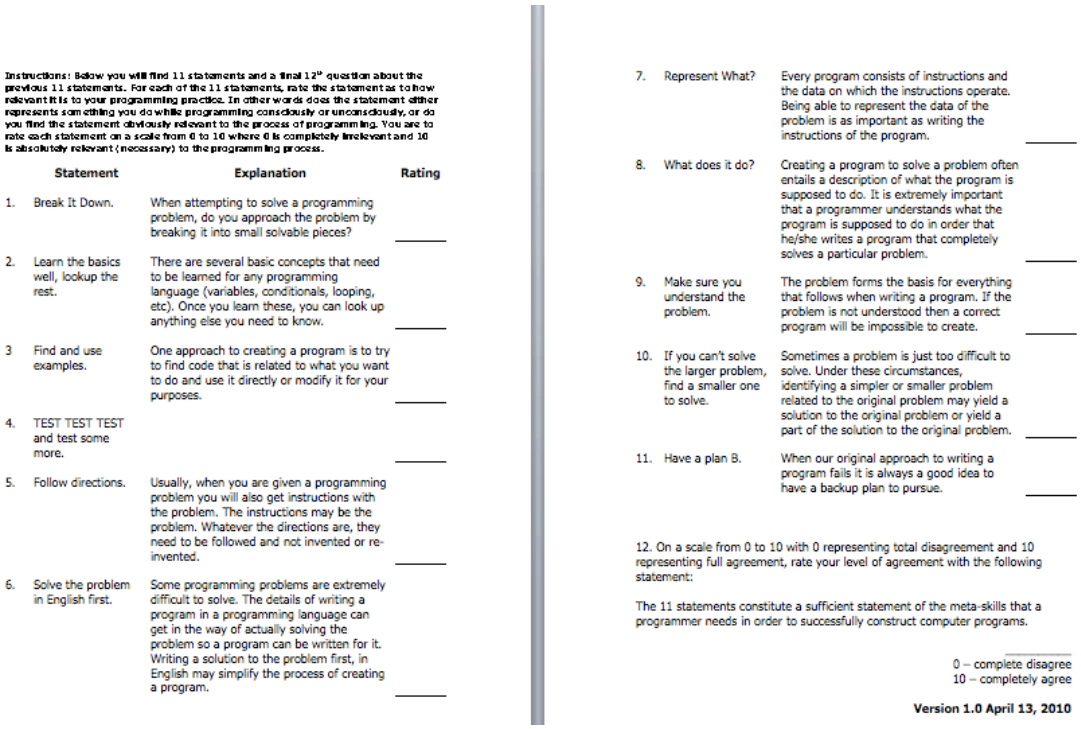
These 11 statements represent a codification of the author's experience in programming and teaching programming. It is extremely easy for students to get lost and frustrated when they first learn about programming. These statements are meant to give the student some strong foundations to stand upon. They address important aspects of the programming process and can be referred to as needed when students are solving programming problems.

The efficacy of these statements can only be measured over a long period of time. We want to know whether knowing these heuristics have the following effects on novice programmers.

> Do the statements result in lesser frustration when learning how to program or solving programming problems?
>
> Do the statements make the programming process clearer?
>
> Do the statements simplify the programming process?
>
> Do you find yourself referring to the statements with any frequency?
>
> Are the statements easy to remember?
>
> Can you apply the statements to all aspects of the programming process?

7. Preliminary Data

To begin the process of analyzing these statements and their relevance to the programming process we asked upper class students in a computer science program to rate the various statements and their relevance to the programming process. The students we asked have been programming for at least two years. Each student had experience programming with at least one programming language. The instrument used for this survey in shown in Figure 2.



Instructions: Below you will find 11 statements and a final 12th question about the previous 11 statements. For each of the 11 statements, rate the statement as to how relevant it is to your programming practice. In other words does the statement either represents something you do while programming consciously or unconsciously, or do you find the statement obviously relevant to the process of programming. You are to rate each statement on a scale from 0 to 10 where 0 is completely irrelevant and 10 is absolutely relevant (necessary) to the programming process.

	Statement	Explanation	Rating
1.	Break It Down.	When attempting to solve a programming problem, do you approach the problem by breaking it into small solvable pieces?	_____
2.	Learn the basics well, lookup the rest.	There are several basic concepts that need to be learned for any programming language (variables, conditionals, looping, etc). Once you learn these, you can look up anything else you need to know.	_____
3	Find and use examples.	One approach to creating a program is to try to find code that is related to what you want to do and use it directly or modify it for your purposes.	_____
4.	TEST TEST TEST and test some more.		_____
5.	Follow directions.	Usually, when you are given a programming problem you will also get instructions with the problem. The instructions may be the problem. Whatever the directions are, they need to be followed and not invented or re-invented.	_____
6.	Solve the problem in English first.	Some programming problems are extremely difficult to solve. The details of writing a program in a programming language can get in the way of actually solving the problem so a program can be written for it. Writing a solution to the problem first, in English may simplify the process of creating a program.	_____
7.	Represent What?	Every program consists of instructions and the data on which the instructions operate. Being able to represent the data of the problem is as important as writing the instructions of the program.	_____
8.	What does it do?	Creating a program to solve a problem often entails a description of what the program is supposed to do. It is extremely important that a programmer understands what the program is supposed to do in order that he/she writes a program that completely solves a particular problem.	_____
9.	Make sure you understand the problem.	The problem forms the basis for everything that follows when writing a program. If the problem is not understood then a correct program will be impossible to create.	_____
10.	If you can't solve the larger problem, find a smaller one to solve.	Sometimes a problem is just too difficult to solve. Under these circumstances, identifying a simpler or smaller problem related to the original problem may yield a solution to the original problem or yield a part of the solution to the original problem.	_____
11.	Have a plan B.	When our original approach to writing a program fails it is always a good idea to have a backup plan to pursue.	_____

12. On a scale from 0 to 10 with 0 representing total disagreement and 10 representing full agreement, rate your level of agreement with the following statement:

The 11 statements constitute a sufficient statement of the meta-skills that a programmer needs in order to successfully construct computer programs.

_____

0 – complete disagree
10 – completely agree

Version 1.0 April 13, 2010

Figure 2 – Survey used to Preliminarily Evaluate Programmer's Wisdom

Each of the statements is shown with a brief explanation. The statements are considered on a scale from 0 to 10 where 0 indicates a statement that has no relevance to the programming process, and where 10 indicates that the statement has significant relevance to the programming process. In addition a 12th question asks the student to rate all 11 statements in terms of their relevance to the programming process. A summary of the ratings given is shown in the next table.

Question		Average	S.D.
1	Break it down	7.9	3.0
2	Learn the basics well, lookup the rest	8.7	3.2
3	Find and use examples	7.6	2.7
4	Test, test, test and test some more	7.8	3.0
5	Follow directions	8.5	1.7
6	Solve the problems in English first	7.1	1.9
7	Represent What?	7.2	1.8
8	What does it do?	8.7	2.0
9	Make sure you understand the problem	8.8	1.1
10	If you can't solve the big problem solve a smaller one	8.1	2.1
11	Have a plan B	7.1	3.0
12		8.9	1.1

Table 1 – Student Wisdom Ratings (N=12)

Looking at this table it is clear that students who have programmed for some time consider the statements representative of the programming process. The lowest average score for a statement was 7.1. Two statements (6 and 11) had this score. The highest score, 8.8, was for statement 9.

8. Conclusion

In order to address the instruction of the necessary meta-skills for the programming process in a concise, memorable, understandable, and usable a series of 11 statements were created. These 11 statements represent a sample of the necessary meta-skills writing computer programs. The 11 statements are not meant to be comprehensive or even complete and one would suspect that over time the statements would possibly evolve or be modified as they are used. This evolution can only be accomplished if they are conveyed to novice students and then considered for their effectiveness.

My primary purpose in this paper is to present these statements as a starting point for others to utilize this formulation of meta-skills. To the extent these statements resonate with other teachers and these teachers use some form of them, the efficacy of these statements can be explored among a wider audience.

The compactness of these statements promotes their presentation in a short period of time – at most two classes. In this time frame, students can learn the wisdom (heuristics), memorize them and apply them to subsequent programming problems.

9. References

Blackwell, A. (2002). What is programming? PPIG 2002, MIT Press.

Brooks, F. P. (1995). The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Addison-Wesley Professional.

Buckingham, B. C., L. Hynd, et al. (2004). "Ways of Experiencing the act of learning to program: A phenomenographic study of introductory programming students at university." Journal of Information Technology Education **3**: 143-160.

Davis, A. M. (1995). 201 Principles of Software Development, McGraw-Hill.

Gries, D. (1974). What should we teach in an introductory programming course? Proceedings of the fourth SIGCSE technical symposium on Computer science education, ACM**:** 81-89.

Mayer, R. E. (1981). "The Psychology of How Novices Learn Computer Programming." ACM Comput. Surv. **13**(1): 121-141.

McConnell, S. (1993). Code Complete, Microsoft Press.

McIver, L. and D. Conway (1996). Seven Deadly Sins of Introductory Programming Language Design. Proceedings of the 1996 International Conference on Software Engineering: Education and Practice, IEEE Computer Society**:** 309.

Polya, G. (1988). How to Solve It A new Aspect of Mathematical Method, Princeton Science Library.

Shaft, T. M. (1995). "Helping Programmers Understand Programs: The Use of Metacognition." Database Advances **26**(4): 25-46.

Sheil, B. A. (1987). The psychological study of programming. Human-computer interaction: a multidisciplinary approach, Morgan Kaufmann Publishers Inc.**:** 165-174.

Shuhidan, S., M. Hamilton, et al. (2009). A Taxonomic Study of Novice Programming Summative Assessment. Eleventh Australasian Computing Education Conference (ACE2009) . Wellington, New Zealand.

Spolsky, J. (2004). Joel on Software, Apress.