Incorporating Debugging into Computer Programming Instruction

By Dr. Randy M. Kaplan
Department of Computer Science
Kutztown University of Pennsylvania
Kutztown, PA

I. Abstract

When a student begins a program of study in computer science or information technology they will be required to take an introductory course in programming. For many students the idea of having to do this is not desirable and this belief is born out when they are taking the course. Computer programming is an extremely complex task that depends on many supporting skills that might also challenge the student (except in a very few cases). In many novice programming courses there seems to be a mismatch between what is taught and what should be taught. This paper will elaborate some of the aspects of missing knowledge that we may provide the beginning student when we teach computer programming to our students. Hopefully if we teach our students more of what they need to know in these preliminary courses, the result will be that we improve the competence of students when they leave their first programming course.

II. Introduction

Consider the following conversation between an instructor and a student taking a first course in Java programming after having completed the necessary prerequisite courses in programming.

```
Student: Hello Professor. I need some help with
my first programming assignment.

Professor: Ok, Jim, what seems to be the
problem?

Jim (student): Well I wrote the first program …
ahh … you know … the one that is supposed to
calculate the area of a circle given the radius
of the circle. I can’t seem to get it to work.

Professor: What seems to be the problem that
you cannot resolve?

Jim: The code compiles just fine but it doesn’t
run correctly. It only prints out zero no
matter what number I set the radius to. I tried
it about a million times.

Professor: Do you have any idea where the
problem might be?

Jim: No. No idea at all.
```

Professor: Do you have any notion of what in the program might be causing this?

Jim: No, I really don't.

Professor: Have you ever written a program similar to this one in your other programming classes?

Jim: Well yes, but that was in C++ and we all got it to work.

Professor: So this program can't be so different.

Jim: Well that's what I mean – I've done it before and I know I am doing the calculation correctly but I still get zero as the answer. There must be something wrong with Java. That's the only thing I can think of.

Although contrived this hypothetical conversation between student and teacher could very well occur. The student, having written a program assignment cannot make it run correctly. They cannot even locate where the error is in the program. The question is how could they ever have passed through the prerequisite courses without knowing how to find program errors? As it turns out many students don't have the faintest idea of how to locate and correct program errors. They have few techniques they can use to find problems in programs they write. For example, many students are not taught to this day how to use simple print statements to debug a program. They seem to see this as a very odd activity when shown. Testing, but more so debugging was not a skill sufficiently reinforced in prior courses (at least it would seem so). When a student cannot find errors and remove them from a program this is a source of tremendous frustration for them. This frustration can lead to a feeling of failure when it comes to programming and may be a factor in changing their program of study in the worst case.

One possibility for resolving this problem is to incorporate more (much more?) about testing and debugging in our first courses. While we are teaching programming we should also be teaching the requisite testing and debugging skills concomitantly. For example, when we assign a programming problem, we should also include a part of the assignment that results in testing/debugging the program. If we are trying to produce better software, why not produce it as close to the beginning of the process as possible as opposed to later in the process. This approach may seem to mimic test-driven programming (Miller, 2004). Test-driven development is a complete software development methodology. The approach suggested here is not a complete methodology. It represents an addition of attention to an important skill whereas test-driven programming represents an approach to creating software. A test-driven programming approach explicitly requires testing (and debugging) to be taught because it is key to the approach. If we look at the way we teach programming, testing and debugging does not reach the status of a first class skill. In this paper we argue for its status as a first class skill.

The discussion about what testing/debugging is and how to approach it should begin with the very first program that a student has to write. If that first program is the "Hello World" program then this approach would mean that we begin speaking about testing and debugging with that program.

III. The Problem

Students new to programming typically don't have a clue of what to do when they encounter their first program error. If the error is a compile time error, the messages are usually less than helpful in giving precise information about the error. One common example is leaving out a closing brace in Java which may elicit the error message, "End of file reached before program's end." When is it that we discuss errors like this one? In my experience in teaching programming we usually teach it when a student comes and asks, why doesn't my program work? Or worse yet, it isn't addressed in the classroom but in the computer lab where other students, usually those who have their programming "battle scars" help students with programs. A problem with this approach is that these battle scarred students usually can rapidly find simple errors and correct them with little or no explanation to the beginning student. As this interaction moves to more complex problems the beginning student does not gain the much needed skills in locating simple program errors. The beginning student will either become "battle scarred" or decide that programming is not for them – the latter happening much too often.

To support this claim we will consider an example from a textbook that uses the "Hello World" program to begin its discussion of Java programming. We arbitrarily choose a text that teaches this program and do not pick on a text for any particular reason. The following is from Horton (2000)[1].

In chapter 1, a section of the chapter entitled "Java Applications" begins as follows,

> *"Every Java application contains a class that defines a method called main(). You can call the class whatever you want …"*

In the next paragraph we get a little closer to the actual program.

> *"We'll see how this works by taking a look at just such a Java program. You need to enter the code using your favorite plain text editor, or if you have a Java development system …"*

The program that the author wants the student to enter is a variation of the "Hello World" program. For our purposes we will show the canonical version.

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //
Display the string.
    }
}
```

After describing the purpose of the various statements of the program, the author instructs the student,

> *"You can compile the program using the JDK compiler with the command …"*

And then,

1 I have used this text in a first course of Java programming.

"Once you have compiled the program successfully, you can execute it with the command …"

What does the author mean by the statement, "*Once you have compiled the program successfully*?" Is it possible to compile the program unsuccessfully? Obviously it must be possible, otherwise why would he say this? So what happens if the program does not compile successfully? From the standpoint of the book this is the reader's problem because the next section of the book is about Java and Unicode.

This is not meant to be any criticism about this text – in fact this approach is consistent with most textbooks that use this program as the first program that is taught. Leaving out how to deal with a compilation error is a large part of the story when first learning to write computer programs.

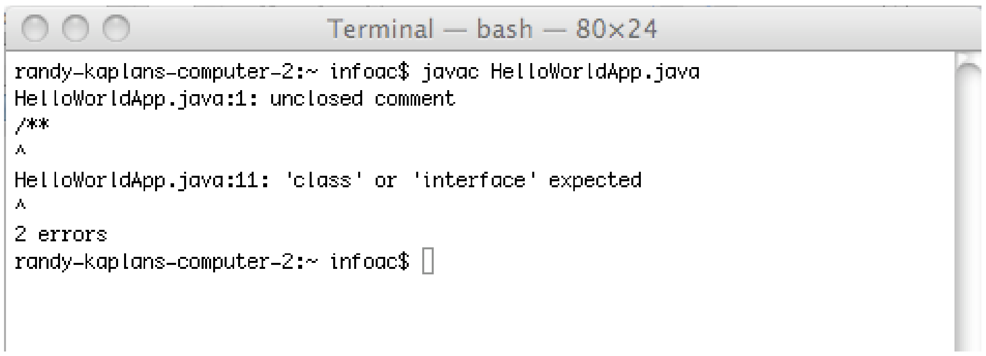
IV. Introducing the Skills of Testing and Debugging as an Integral Part of the Programming Process

Teaching a programming language to any student today involves the conveyance of a significant number of subject-independent and subject-dependent skills. Programming is fundamentally a problem-solving process in a specialized domain. The process involves many different skills and in order to be a successful student of programming the sooner that these skills can be brought to bear by the student, the sooner they will be successful in their programming activities. There has been significant research into these skills that date back almost to the beginning of when programming reached some level of public visibility. Specifically this research has examined the nature of the knowledge needed and the cognitive skills needed to be a programmer. For example Deek and McHugh (2003) delineate six knowledge areas that are required to produce a successful computer program. These include knowledge about problem formulation, solution planning, solution design, solution translation, solution testing, and solution delivery.

Some of the results of this research has been brought to bear on approaches to teaching programming, but amazingly enough, the focus on specific bodies of knowledge and how they interplay with the programming skill has not been examined.

The teaching of writing computer programs to students with little or no experience in doing this should approach the topic from the standpoint of not only how to write programs but also incorporates knowledge about testing, debugging, and in general, problem solving. The idea behind this is that students become stuck when learning the initial stages of the programming process because (a) they don't know what to look for (we don't explicitly tell them) and (b) they don't know where to find the information they need to get "unstuck." This empirical description of the problems that novice programmers seem to have when learning to write computer programs is confirmed by various multinational studies that have been carried out. The McCracken Study (McCracken et al, 2001), and a second study carried out by Lister et al (2004) both indicate that a significant number of students at the end of their first year of a program in computer science do not have the requisite skills to be successful.

Consider the following example and the kind of error that would be displayed, if for example, the terminating "`*/`" was left off of the comment block. What would happen? The error that occurs when compiling the code as described produced the following



```
randy-kaplans-computer-2:~ infoac$ javac HelloWorldApp.java
HelloWorldApp.java:1: unclosed comment
/**
^
HelloWorldApp.java:11: 'class' or 'interface' expected
^
2 errors
randy-kaplans-computer-2:~ infoac$ 
```

The errors, "/**" and "'class' or 'interface' expected don't really tell us too much about what caused the problem. There is in fact a class declaration, and what does the bit about "/**" mean? Many compilers still output obtuse messages that seem to have nothing to do with the actual problem. What is the student to do? Where should they look for the error? When does such an error occur? In the next section we discuss the details of what may be done in situations like this one and similar ones in order to get the student quickly through the correction of the error with an understanding of why the error was caused thus beginning the training of awareness that is required to specific details of the programming process like debugging.

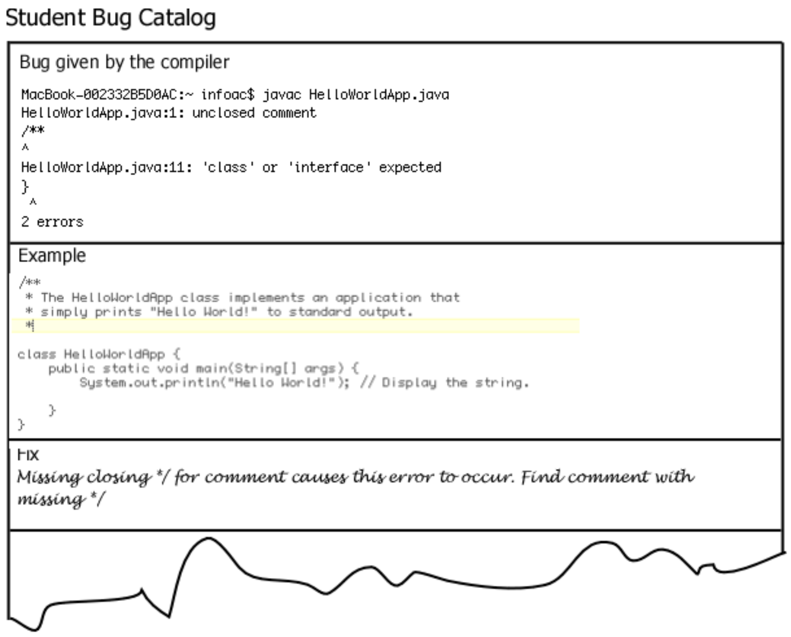
V. The Details

Since we are trying to promote more of the skills necessary to carry out the programming task from beginning to end we suggest a series of exercises that explicitly deal with aspects of programming relevant to the debugging task. We distinguish between debugging and testing at this time because we consider testing the process of exercising a program to discover potential programming bugs and debugging the process of removing the bug from the program. The exercises we suggest will be explicitly about the recognition of bugs and what to do about them. The two major classes of bugs we will be concerned with are compile-time errors and run-time errors. For run-time errors we will consider the basic kinds of errors that could arise since there are almost an infinite number of run-time errors that may arise. Fortunately most of these in the beginning come under what we would call "basic" which allows us to propose a number of relevant exercises.

Compile-Time Errors

When a student encounters a compile-time error it is often the case that they do not know why the program did not compile, and unfortunately, how to relate the error to the actual program code they have written. For this reason it would make sense for the student to create a catalog for themselves of the kind of errors they may encounter and also instructions as to how to fix a particular kind of error. In order to aid in the student's compilation of such a catalog we start the student off with a program like the canonical "Hello World" program by giving them the exact program and asking them to create a file with the program in it. There is a relatively good chance at this point, that unless they are extremely careful when create the program file, that they will receive their first compile time errors. If such errors are received at this point we ask the student to carry out the very first exercise – "Code Comparison."

In the code comparison exercise we ask the student to compare each and every character that they entered to each and every character in the original program. In doing so it is hopeful that they will locate a difference and correct the difference in the program file and then try to compile the program again. In addition to correcting the particular error we also ask the student to begin keeping their catalog of potential errors that can appear at compile-time. We not only suggest that students maintain the catalog but also suggest a particular format for the catalog so that they may easily refer back to the catalog when an error occurs. The format for an entry in the catalog is shown in the next figure.


Student Bug Catalog

Bug given by the compiler

```
MacBook-002332B5D0AC:~ infoac$ javac HelloWorldApp.java
HelloWorldApp.java:1: unclosed comment
/**
^
HelloWorldApp.java:11: 'class' or 'interface' expected
}
 ^
2 errors
```

Example

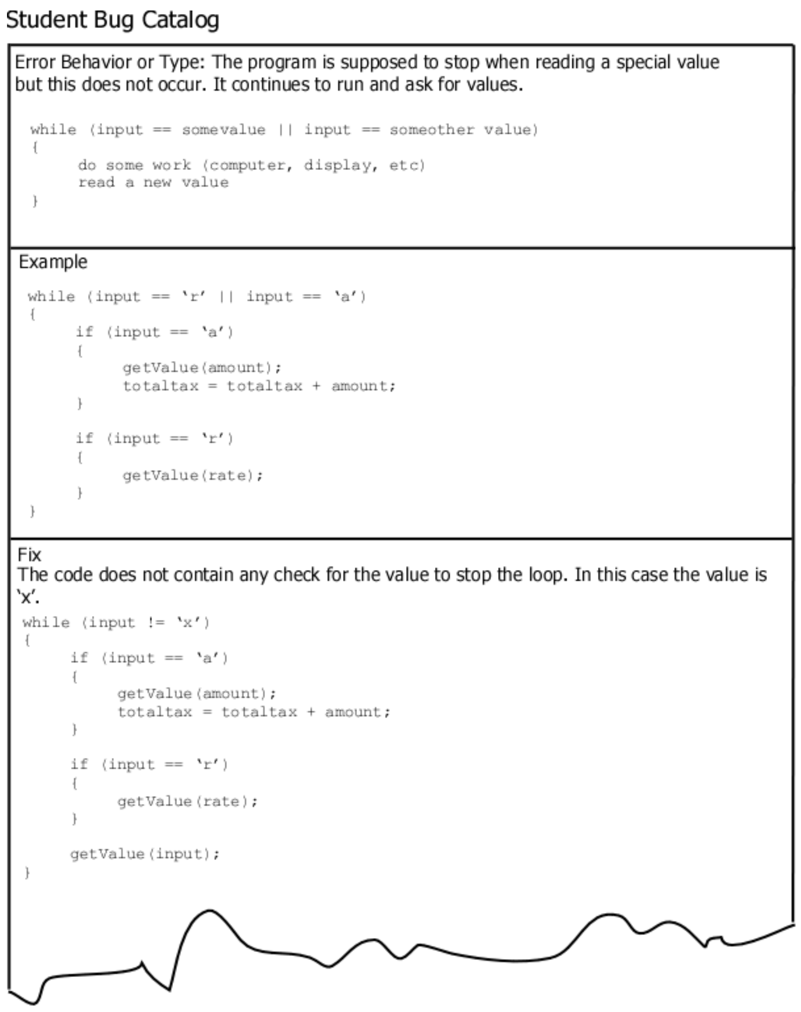
```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 *

class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.

    }
}
```

Fix

*Missing closing */ for comment causes this error to occur. Find comment with missing */*

Compile-time errors represent the simpler class of the kinds of errors that students would encounter when writing their first programs. The more complex kinds of errors, which are more difficult to find, are the run-time errors that students will face once they are able to get a program to compile. Student comments from McCracken's study (2001) clearly indicates from that students have little or no idea of what to do when they must correct run-time errors. Specifically, a student must be able to determine why a program they write does not produce the desired result (output). Pea (1987) conjectures that such errors may be caused by assumptions made in complex reasoning processes. They describe an example of a reasoning process that involves goals, goal decomposition, and then, after translation into the computer program, an incorrect merging of goals so that in essence they think they have written a correct program (one that meets the original goals) but they have not. It is my belief that this type of error (and those related to it) is fundamental to the success of any novice programmer.

If we consider our catalog to contain a comprehensive list of the types of problems that might be encountered during the programming process, these types of errors must also be addressed. For the sake of an example, we adopt one of Pea's examples to the Java programming language and show what the entry in the student bug catalog might look like.



Student Bug Catalog

Error Behavior or Type: The program is supposed to stop when reading a special value but this does not occur. It continues to run and ask for values.

```
while (input == somevalue || input == someother value)
{
      do some work (computer, display, etc)
      read a new value
}
```

Example

```
while (input == 'r' || input == 'a')
{
      if (input == 'a')
      {
            getValue(amount);
            totaltax = totaltax + amount;
      }

      if (input == 'r')
      {
            getValue(rate);
      }
}
```

Fix
The code does not contain any check for the value to stop the loop. In this case the value is 'x'.

```
while (input != 'x')
{
      if (input == 'a')
      {
            getValue(amount);
            totaltax = totaltax + amount;
      }

      if (input == 'r')
      {
            getValue(rate);
      }

      getValue(input);
}
```

This example presumes that the student is able to abstract the code shown and apply it to their specific case and problem. As this author has seen, students may not be able to do that so at some point students must be explicitly instructed as to how this can be done.

What benefits may students gain from having a catalog like the one we are suggesting? Since the catalog calls attention to the kinds and types of errors that might be encountered by a student, they would at least have some resource to turn to so that they may attempt to solve a problem they are having on their own. The very nature of using the catalog to solve a problem represents a reinforcement of problem solving behavior.

Secondarily, for the teacher the bug catalog represents a resource they would be able to teach from. If they give a programming assignment they would be able to show, using the catalog, the kinds of errors that would arise when carrying out the assignment. In this way the teacher is directly addressing problems that may come up and showing them how to solve them. The catalog then becomes the reminder of the instruction given by the teacher.

Ultimately we would like to see textbooks about programming take a more knowledge-based and problem solving approach. The first topic in such a textbook might be a general overview of problem solving without relationship to programming, and the next topic could relate problem solving directly to programming. Specific attention could then be given to the kinds of knowledge that a students needs to have in order to solve programming problems.

VI. Related Work

Wilson (1987) identifies a simple approach to teaching novices learning a new programming language how to identify bugs in a program they have written. The approach, using a variant of Socratic discourse, forces a student to describe their solution to a programming problem without using the code. The goal of making a novice do this is to get them to build a mental model of their problem solution. By doing this they are able to gain insight into what might be wrong with the solution to the problem.

Much of the related work in teaching students debugging takes the approach of identifying the problems that novice programmers have when they are writing programs. One of the more interesting findings was that students at the end of their first programming were not able to describe the workings of their own programs. Three recent studies, McCracken (2001), Lister (2004), and Tenenberg (2005) all show this same result. After completing the first programming students, the majority of students could not explain or design simple programs.

Pea et al (1987) in a study of programs written by novice programmers delineate a number of specific errors that are made when students are writing programs. There are two important results in this work. First, we see the beginnings of a hierarchy of program errors that may be made. Although Pea and his colleagues do not couch their delineation in these terms, there is definitely some attempt to classify the errors they describe. These errors unlike some of the others we have described represent complex reasoning issues that should be addressed when teaching a student to program. They state that most instruction in programming seems to consist of a syntax-based approach focusing on the programming language being taught as opposed to teaching problem-solving skills that are key to the programming process. These two results alone represent a basis for how we might approach teaching programming to beginning computer science students.

VII. Conclusions and Further Work

In this paper we have highlighted the need for additional aid for the beginning student of computer programming. Based on past research it has been shown that when a student leaves the first course in programming, that for the most part they are unable to successfully write a working program. Only a small percentage of students are able to do this leaving the majority unable to utilize computers for even the simplest applications. Historically programming has been taught from the standpoint of learning the programming language, i.e., how to write statements in the language. To a large degree the semantic and pragmatic considerations of the language have been left to the student to learn. Some magic form of osmosis occurs and the student knows how to problem solve in this domain. Given that research has shown that most students do not experience the magic, it behooves us to given students resources to use to be successful at this skill. One such resource would be a reference elucidating the kinds of problems and the kinds of errors they would encounter. A second solution/resource would be to change the textbooks about programming to incorporate knowledge about the process in addition to knowledge about the programming language. Finally, consideration should be given to how programming is taught in the classroom. A more problem-driven approach may result in an increase in student competence in programming.

References

Deek, F. P. and J. McHugh (2003). PROBLEM SOLVING AND COGNITIVE FOUNDATIONS FOR PROGRAM DEVELOPMENT: AN INTEGRATED MODEL Sixth International Conference on Computer Based Learning in Science. Nicosia, Cypress.

Horton, I. (2000). Beginning Java 2. Chicago, Wrox.

Lister, R., E. S. Adams, et al. (2004). A multi-national study of reading and tracing skills in novice programmers. Working group reports from ITiCSE on Innovation and technology in computer science education. Leeds, United Kingdom, ACM.

McCracken, M., V. Almstrum, et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. Working group reports from ITiCSE on Innovation and technology in computer science education. Canterbury, UK, ACM.

Miller, K. W. (2004). "Test driven development on the cheap: text files and explicit scaffolding." J. Comput. Small Coll. 20(2): 181-189.

Pea, R. D. and D. M. Kurland (1983). On the Cognitive Prerequisites of Learning Computer Programming. New York, Bank Street College of Education.

Tenenberg, J. and e. al (2005). "A multi-national, multi-institutional study." Informatics in Education 4(1): 143-162.

Wilson, J. D. (1987). "A Socratic approach to helping novice programmers debug programs." SIGCSE Bull. 19(1): 179-182.