Teaching Tip
Teaching Introductory Programming from A to Z: Twenty-Six Tips from the Trenches

Xihui "Paul" Zhang
John D. Crabtree
Mark G. Terwilliger
Janet T. Jenkins
Department of Computer Science & Information Systems
College of Business
University of North Alabama
Florence, AL 35632, USA
xzhang6@una.edu, jcrabtree@una.edu, mterwilliger@una.edu, jltruitt@una.edu

ABSTRACT

A solid foundation in computer programming is critical for students to succeed in advanced computing courses, but teaching such an introductory course is challenging. Therefore, it is important to develop better approaches in order to improve teaching effectiveness and enhance student learning. In this paper, we present 26 tips for teaching introductory programming drawn from the experiences of four well-qualified college professors. It is our hope that our peers can pick up some tips from this paper, apply them in their own classroom, improve their teaching effectiveness, and ultimately enhance student learning.

Keywords: Teaching tip, Introductory programming, Teaching effectiveness, Student learning

1. INTRODUCTION

An introductory programming course is usually a requirement for lower-level college students who are majoring in Computer Science or Information Systems. Because content taught at this level is so fundamental, students who struggle with it will inevitably struggle in advanced computing courses. In a sense, the introductory programming course serves as a gateway course. Teaching the introductory programming course is also complicated by the variety of backgrounds that the students present: some come into the class with substantial experience, while others have little to none. Consequently, it is important to develop approaches that will improve teaching effectiveness, which in turn will enhance student learning.

Millions of dollars have been spent from the grant and award funding agencies such as the National Science Foundation and by universities on different strategies such as computer labs, supplemental instruction, and tutors in an effort to find better ways to teach introductory programming. Realistically, however, this is not an easy task. It is akin to asking someone who has never built a house, and who does not speak French, to build a house in France using plans that were written in French. Programming requires a number of mental skills, and the required skills are similar to those used in advanced mathematical reasoning. The Common Core State Standards for Mathematical Practice (http://www.corestandards.org/Math/Practice/) are:

- Make sense of problems and persevere in solving them.
- Reason abstractly and quantitatively.
- Construct viable arguments and critique the reasoning of others.
- Model with mathematics.
- Use appropriate tools strategically.
- Attend to precision.
- Look for and make use of structure.
- Look for and express regularity in repeated reasoning.

Anyone who has ever attempted to teach introductory programming to students will recognize that the majority of skills on this list apply to programming as well. To help students achieve these learning outcomes, we must present the curriculum in a manner that gives the students opportunities to build these skills. In addition, we need to provide a supportive learning environment in and out of the classroom. A multi-faceted approach is required to successfully lead a group of students through an introductory programming course.

In this paper, we developed a list of 26 tips. We believe these tips could be adopted by others to improve their teaching effectiveness and attain improved student learning outcomes.

2. METHODS

To compose these tips for teaching introductory programming, we assembled a team of four faculty members who have taught or are actively engaged in teaching such courses. Each faculty member has had not only extensive experience in teaching introductory programming, but also extensive industry work experience in programming or software development. The background of the team members and the process of developing the teaching tips are detailed as follows.

2.1 Team of Four

All four faculty members are presently professors of computer science or computer information systems at a university located in the mid-south region of the United States. At the current school, we have a total of 51 years of teaching experience. We offer two similar introductory programming courses, one for Computer Information Systems (CIS) majors and another for Computer Science (CS) majors. Two of the co-authors teach sections of the CIS introductory programming course, while the other two teach sections of the CS introductory programming course. Additionally, the four faculty members taught programming and computing-related courses for a combined 36 years at six other institutions. Furthermore, the four faculty members have 27 combined years of industry experience in programming-related careers or in the field of software development.

2.2 The Three-Step Process

We followed a three-step process when developing the final list of our teaching tips. Each member of the team independently created a set of teaching tips. For each tip, we made an effort to ensure that the tip was both clear and concise, but included sufficient details and examples for illustrative purposes. These tips were then compiled into a single list, with overlapping tips consolidated. Finally, we focused on determining how best to present the final list.

Independently, we came up with a total of 32 teaching tips; after consolidation, we reduced this list to 26. We then labeled each tip with a distinct verb and chose to list the resulting tips in alphabetical order.

3. TWENTY-SIX TEACHING TIPS

3.1 Assign Homework on Debugging

One of the most frustrating obstacles to new programmers is when they hit a dead end and cannot figure out a way to debug their code. Debugging is an important skill that continues to be both difficult for novice programmers to learn and challenging for computer science and information systems educators to teach (McCauley et al., 2008). Giving students explicit early practice with debugging is a way to develop skills that may come in handy later. Program errors typically come in three categories: (1) syntax errors – the code has violated a language rule (undeclared variable, punctuation issue, etc.), (2) run-time errors – the program "crashes" because of an unexpected situation (division by zero, invalid memory access, the dreaded infinite loop, etc.), and (3) logical errors – the program is producing the wrong output (instructions out of sequence, wrong formula, etc.).

A debugging homework problem should be assigned very early in the semester in which students are given several programs that each contains multiple syntax errors. This allows students to become familiar with the compiler's error messages. Students have to correct each error and then add comments to the programs that describe the errors they correct. Additional assignments may be given later in the semester that have run-time and logical errors. Common examples that trip up students who are learning languages like C++ or Java are using "=" instead of "==" for comparison and including a semi-colon after the test of the condition of a *while* loop.

3.2 Begin with an Exciting Application

Ever since the publication of Kernighan and Ritchie's book (*The C Programming Language*), we have been tacitly obliged to start our students with a "hello, world" program. However, such programs rarely motivate or excite students and even were considered harmful (Westfall, 2001). A significant program with audio and/or visual elements can accomplish several pedagogical goals. First of all, it can display what well-written code, complete with informative comments, should look like. Secondly, it can be an opportunity to point out that such a result can rarely, if ever, be achieved without significant thought and planning (i.e., design). Lastly, such a program, if chosen carefully, can be an application that serves as a goal for the student upon completion of the course.

We have used the BlueJ development environment and a textbook authored by Barnes and Kölling (2016) to introduce the Java programming language. The textbook contains several simple graphics programs that students can execute and modify on the first day of class. We invited the students to run a program that displays a red house with the sun above it. After asking them to peruse the source code, we challenged them to change the color of the house or add a second sun. This code includes six well-written Java classes.

Now that we are using Python to teach the same course, we begin the first class by having students execute a pygame application in which they shoot aliens. We ask them to change the background color, slow the game down, or add their names to the title of the game. They are exposed to well-written Python code and are generally surprised at how easy it is to manipulate the code and see immediate results.

3.3 Code Early and Often

Students need an opportunity to practice each new concept, whether they are dealing with language syntax and semantics or problem solving concepts and algorithms. They need to be able to solve problems themselves, not just watch the professor. This can be achieved in a variety of ways. Interweaving lectures, class discussions, and in-class activities allows the students to try out new concepts early in the course. Introducing new programming concepts via live coding exercises can especially benefit those who learn best from a "hands-on" approach (Tan, Ting, and Ling, 2009). Demonstrating these concepts with this approach (by the instructor, the student, or both) can assist in making the concepts clear and memorable.

This approach can be used when teaching many of the basic concepts. Introductory programming courses often focus on the three critical components of any complete program: input, processing, and output. Writing a complete, working

program in class can commence with the students following along while the outline of the program is written by the instructor. Students can then be challenged with one of the steps of a basic program (e.g., writing the code to accept user input). This can give the instructor the opportunity, in a small classroom, to move about the room and help those who may be struggling as well as to provide the opportunity to reinforce the input, processing, and output model. Verbal instructions can be given to those who have finished the first task while the instructor continues to help others. As students progress, the instructor can return to the front of the classroom and continue developing the program. A whiteboard is invaluable in assisting with concepts or visual depictions of the internal working of the code or the design of the solution in terms of a flowchart or pseudocode. Having students write their own code, along with the instructor-led coding and instructor-led explanations on the whiteboard, provides a variety of mental activities so students with different learning styles are able to assimilate the input, processing, and output model into their programming skill set.

Smaller programming assignments frequently given between class meetings allow the students to have ample opportunities for learning outside of class by working out a problem with newly learned concepts. A larger programming assignment typically gives students a better and more challenging opportunity to design and implement a software solution. These activities provide a variety of opportunities for students to learn by doing from the onset of the course and allow the professor to recognize problem areas.

3.4 Design and Code Exercises with Published Solutions

It can be very helpful to encourage students to work problems that are presented in the textbook along with the author's solution code, which is usually provided as a supplement or in an appendix. Students will benefit most by designing and, if time allows, coding their own solution before looking at the author's code.

However, since most students will not have time to code each and every program in the book, the instructor can encourage students to at least design a candidate solution in their heads, if not on paper, before looking at the author's solution. This is an excellent opportunity to emphasize coding as a creative endeavor. It is doubtful that the students will approach the problem in the same way as the author, but they may have a correct solution derived in a unique fashion – one of the many enjoyable aspects of programming.

This activity may also prompt a discussion of evaluating the elegance of a particular approach. Although there are often many valid solutions to a given programming exercise, not all solutions are qualitatively the same. Some may minimize the amount of code written. Verbose solutions, while requiring more lines of code, may be easier to read. A discussion of the trade-offs involved and the notion of "elegant" solutions may prove valuable to beginners.

3.5 Emphasize Code Style and Demonstrate Conventional Structures

Clean, well-organized, and consistently formatted code improves the code's readability, decreases the chances of making errors in the code, and makes the code easier to maintain. This is more critical when the code is maintained by people other than the original programmer. As such, it is important to emphasize code style to students in introductory programming courses. Consistency is the most important measure for code format. Indentation, spacing, code blocks, class member ordering, maximum line length, and parentheses should all be used consistently. Even though most of today's IDEs (integrated development environments) provide functionality to format the source code automatically, it is still important to emphasize to students why the code style is important and demonstrate how to format the code.

For a structured program, it is important to demonstrate the Input-Processing-Output structure to students. A novice programmer will typically mix up the processing and output, especially when there are multiple items to process and the output contains multiple lines. For instance, in the example provided in Figure 1, some novice programmers will print the first output message once the *max* value is found, and then move on to find the *min* value and print the second output message. This will change the Input-Processing-Output structure into an Input-Processing-Output-Processing-Output structure. A better coding practice is to find both the *max* value and the *min* value first, and then print the two output messages, as shown in Figure 1.

```
import java.util.Scanner;

public class MaxMin {
    public static void main(String[] args) {
        // Input
        Scanner input = new Scanner(System.in);
        System.out.print("Enter three different integers: ");
        int a = input.nextInt();
        int b = input.nextInt();
        int c = input.nextInt();

        // Processing
        int max = a; // largest of a, b, and c
        if (max < b) {
            max = b;
        }
        if (max < c) {
            max = c;
        }

        int min = a; // smallest of a, b, and c
        if (min > b) {
            min = b;
        }
        if (min > c) {
            min = c;
        }

        // Output
        System.out.printf("The largest number is %d.\n",
max);
        System.out.printf("The smallest number is %d.",
min);
    }
}
```

Figure 1. An Example of the Input-Processing-Output Structure

For an object-oriented program, it is important to emphasize the ordering of the class members when defining a class. The code within a class definition should be grouped into three sections in sequential order, as shown in Figure 2. The first section contains the data fields, the second section contains the constructors, and the third section contains the methods (i.e., getters/accessors and setters/mutators).

```
public class Employee {
    // Data fields
    private String firstName;
    private String lastName;
    private double monthlySalary;

    // Constructors
    public Employee() {
    }

    public Employee(String fName, String lName, double
mSalary) {
        firstName = fName;
        lastName = lName;
        monthlySalary = mSalary;
    }

    // Methods
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String newFirstName) {
        firstName = newFirstName;
    }

    public String getLasttName() {
        return lastName;
    }

    public void setLastName(String newLastName) {
        lastName = newLastName;
    }

    public double getMonthlySalary() {
        return monthlySalary;
    }

    public void setMonthlySalary(double
newMonthlySalary) {
        monthlySalary = newMonthlySalary;
    }
}
```

Figure 2. Three Sections in a Class Definition

Emphasizing the ordering of the class members when defining a class becomes especially important if we adopt an objects-first teaching approach. Some introductory programming languages do not support classes, but emphasizing style conventions is still important. For instance, when using C++ for procedural programming, it is a convention to list function declarations above the *main()* function and function definitions below the *main()* function. In larger projects, the function declarations could be included in *.h* files where the implementation would be located in *.cpp* files.

3.6 Flip the Classroom and Let Students Take Control

In recent years, "flipping the classroom" has become a catchphrase, and the flipped classroom model has been gaining popularity. In a flipped classroom, according to Mok (2014), the instructor "delivers" lectures before class in the form of pre-recorded videos and spends the actual class time engaging students in learning activities that involve collaboration and interaction. The most significant advantage of the flipped classroom model is that it promotes student-centered learning and collaboration. Student-centered, active learning and participation help students better understand challenging programming concepts, render pedagogical benefits, and improve overall teaching effectiveness and learning efficiency (e.g., Benander and Benander, 2008). In an exploratory empirical study involving two sections of an introductory programming course taught by the same instructor in the same semester, Crabtree, Nickels, and Parris (2013) found that teaching sections of lectures interspersed with hands-on programming examples resulted in higher exam scores and a higher student retention rate than those of lectures first and hands-on programming examples second. Furthermore, Zhang et al. (2013) compared the effects of the two teaching approaches on learning performance – the instructor-centric lecture and exercise approach versus the student-centric exercise only approach. Their results support the conclusion suggesting that when teaching introductory programming courses, instructors may want to consider choosing student-centered, active learning over the traditional lecture format in order to gain better student learning performance.

One way that we implement the flipped classroom model is to assign a chapter quiz before the class begins on that specific chapter and spend the actual class time leading the students to work on the programming assignments, interactively and collaboratively. Typically, a chapter quiz contains 10 multiple-choice questions. These questions are content-based and randomly drawn from a test bank provided by the textbook publisher as part of the instructor's resources. The chapter quiz is delivered via Canvas, a trusted, open-source, modern learning management system (LMS) with numerous robust features that support a deep focus on teaching and learning. There are many options to set up the quiz. We typically adopt the following settings for the quiz: (1) shuffle answers, (2) set 30 minutes (about 3 minutes per question) as the time limit, (3) allow 3 attempts and keep the highest as the quiz score, (4) let students see their quiz responses only once after each attempt, (5) show one question at a time, (6) require an access code, and (7) set the due date at midnight the day before the class meeting time.

3.7 Gauge Student Concept Mastery using Exit Tickets

An instructor covering a topic does not necessarily mean students have mastered that concept. Computer programming is, in many ways, similar to mathematics in that most topics are based on previous building blocks. You cannot work with arrays, for example, if you do not understand repetition structures as well as operations with simple data types.

Using exit tickets is a technique that can gauge whether students have mastered a new concept. Exit tickets offer easy, quick, and informative assessments that help encourage student connections to content, self-reflection, and a purpose for future learning (Angelo and Cross, 1993; Marzano, 2012). An exit ticket is literally a small ticket (3x5 index cards work well) that a student must submit in order to leave the classroom. Typically, these are distributed to students with about five minutes remaining in class, and each consists of a short question or two.

The questions could be in a variety of formats. You may ask students to write on paper the lines of code necessary to solve a problem ("Write the code to print all prime numbers less than 100"). You may ask a short concept question ("What is the difference between an array and a class?"). You could ask them to rate how well they followed the day's lecture on a scale from 1 to 10, with 1 being "completely lost" and 10 being "completely understood it." You could ask them about the difficulty level of the assigned textbook readings. The basic idea is that you get immediate feedback on a topic related to the students' learning. Whether you require students to include their names on the exit tickets or if you include their responses as part of their course grade is completely up to you.

3.8 Help Students Build Computational Thinking Skills

Critical thinking skills are imperative for problem solving. In computing, these critical thinking skills fall under the realm of computational thinking (Kules, 2016). Shein (2014, p. 16) maintains that "not everyone needs coding skills, but learning how to think like a programmer can be useful." This is because "computational thinking helps people learn how to think abstractly and pull a problem apart into smaller pieces" (Shein, 2014, p. 17). We are not merely teaching students to "code," we are teaching them to solve problems, and we are teaching them to think computationally.

For example, if a student is asked to determine the minimum value from a list of numbers, the student will scan the numbers, find the minimum value, and declare it. If the same novice programmer is asked to design an algorithm to determine the minimum value in a list of numbers, the novice might create an algorithm that is similar to this: (1) scan all of the numbers; (2) output the minimum. Students may not realize that "scan all of the numbers" is what needs to be detailed. It is helpful to guide the students through an interactive exercise where they are led to see the constraints of the problem.

Ask a student to listen to the numbers called out and, at the end of the list, ask the student for the minimum value. When this happens, the student actually goes through the algorithm that is required to solve this problem. The first number spoken is initially the minimum. Then, they listen to the next number. If the new number is smaller than the minimum, they toss the old minimum and keep up with the new minimum. They repeat this process until no more numbers are given. Whatever number is stored in their brain will be the minimum. The professor can then write the algorithm, draw the flow chart, and develop the actual code needed to solve the problem just mimicked. Students can see how to transform a manual solution for a problem into a computer program.

3.9 Inject Peer Tutors into the Classroom

A couple of years ago, we started having our upperclass peer tutors attend the course lecture sessions in our CS I course. With a classroom of 30 students, each with a computer, this provided opportunities to expand classroom activities. Having another person to assist students made it more manageable to add more essential, hands-on, in-class exercises where immediate feedback was available. Attending class allowed the tutor to deepen his or her understanding of the material and to know exactly what the students had been taught during class. A noticeable difference in the classroom atmosphere was obvious.

In addition to improvements in the classroom environment, there were several unexpected results. The usage of our out-of-class, drop-in tutoring increased dramatically. We believe students became familiar and comfortable with the tutors in the classroom and felt less threatened to visit the drop-in tutoring on their own. We also noticed an observable change in our tutors, as they would frequently stop by our offices to discuss student issues, offer alternative strategies to solving problems, and suggest activities to try in upcoming classes.

Paying student tutors to attend course lectures costs money, but we believe the benefits to the classroom atmosphere and the improved students' mastery of concepts are worth the investment. The unexpected benefits to the tutors acting as near-peer mentors are also a positive side effect (Dickson, 2011; Trujillo et al., 2015).

3.10 Just Go Agile and Team Students Up

Agile software development methods are basically iterative development approaches that focus on incremental specification, design, and implementation (Sommerville, 2016). Agile methodologies are designed to produce high quality software in a cost effective and timely manner, while adapting to meet the changing needs of the end users (Zhang et al., 2010). As a core practice of eXtreme Programming (XP), an early agile methodology, pair programming involves two programmers sitting side by side, sharing a single computer screen, and working on the same software program collaboratively. Going agile and teaming students up will provide them the opportunity to understand and apply agile software development practices. Teamed students will be able to bounce questions and ideas between each other.

In introductory programming courses, the programming ability levels of incoming students can vary greatly. The challenge for the professor then becomes how to give enough detail and instruction to those who have little to no experience without losing the interest of those who have programmed before. As such, we have used an active and cooperative learning approach where teaming is used in the classroom. This is akin to pair programming in agile development, but is more of a pair (or triple) learning situation.

The students are placed in groups of two or three. They are instructed on how to operate in this learning environment. One person (the driver) is to be typing in source code while the teammates (observers/navigators) are reviewing code as it is typed, looking up resources online or from the text, providing feedback, and asking questions of what is being typed by the driver. Teams are composed so that those with experience can share their knowledge with more novice peers. This collaborative work on exercises and the interaction

among students will provide immediate feedback to the professor without needing to visit every student. The professor can survey the room, see how each group is progressing, and gain valuable information about what has been learned or what needs reinforcing.

Many studies have shown teaming students up to be an effective way of learning to program (e.g., Radermacher and Walia, 2011). The research findings by Chen and Rea (2018) suggest that students' problem solving skills and solution formation experience are more important than their prior specific domain knowledge, and that gender and major composition (computing vs. non-computing majors) are important factors to consider when assigning programming pairs. As such, assigning pairs should not be done randomly. Many factors (e.g., the student's gender, major, problem solving skills, solution formation experience, and personality) need to be taken into consideration. Linden (2018) reflected on the implementation of Scrum to create a teaching and learning environment in an introductory programming course that fosters self-regulated learning in students. The evaluation of the Scrum-based learning environment revealed that students want to be in control of their learning.

Learning to program in pairs or triples in class can be beneficial to students. However, novices also need ample opportunity to solve problems and program on their own. They need to know that getting too much help from others is not beneficial to them in the long run. Cheating can also diminish the quality of a program if it becomes widespread (Sheard et al., 2017). Requiring students to do a significant amount of programming work on their own and holding them accountable for doing such work are good for them and good for any academic computing program.

3.11 Keep It Simple

There are so many new things to master when learning to program for the first time. As such, it is important to avoid extraneous components that are not critically related to the curriculum.

3.11.1 Keep it simple (environment). Advanced Integrated Development Environments (IDEs) are great tools for managing large, multi-class, multi-component projects, but they often include extra files and folder hierarchies that can be confusing to novices. A simple IDE, such as Dev C++, which includes a text editor that has settings to correctly assist with proper indentation of code structure, works nicely for beginning programmers. As students move on to second and third programming courses, more advanced IDEs, such as Visual Studio, which include project management tools, can be introduced.

3.11.2 Keep it simple (logic). Often a solution to a problem has multiple layers of logic. When novices are learning to write nested loops, for example, it is useful to show them how to work both inside out and outside in. It may also be helpful to draw flowcharts of more complicated looping structures so students are able to model the code in multiple ways. A basic selection sort is an excellent algorithm for illustrating this process. Students could be asked to design code that will select the smallest (or largest) value in an array segment and swap that value with the value in the first position. The instructor can then show them how to embed that solution in an outer loop that will sort the entire array. The instructor could then begin with the outer loop and then develop the logic for implementing the inner loop.

3.11.3 Keep it simple (compilation). It is important when students are faced with a complex programming problem to break the problem down into smaller parts. If they successfully build a small chunk of code, compile it, debug it, and test it, then they have a working component. This gives them a starting point for the next layer of the solution. This concept matches iterative life cycles in the professional world. Postponing compilation until the entire solution has been coded typically results in more compiler and logic errors than can be handled, leading to frustration and failure.

3.12 Learn from Our Peers and Get Student Feedback

No matter how well we have designed and delivered our course, there is always room for us to improve our teaching and students' learning effectiveness, which can be accomplished through two major approaches. The first approach is to learn from our peers, and the second approach is to get student feedback.

Learning from our peers is very helpful. There are so many ways to learn from our peers. For instance, we can audit a class taught by a colleague who has just won a teaching award. We can talk to senior scholars in our field and ask them for effective teaching advice. We can also attend education conferences such as SIGCSE and EDSIGCON to hear what other professors are doing or read the pertinent conference proceedings. All of these strategies can provide a wealth of fresh ideas to improve our teaching and students' learning effectiveness.

Getting student feedback on course design and delivery is also very important. This feedback is from the student to the instructor, and the majority of it should be constructive, to the point, and actionable. With this feedback, we should be able to develop additional innovative ways to improve the design and delivery of our introductory programming course. This, in turn, can help improve our teaching and students' learning effectiveness.

3.13 Maintain a Steady Rhythm

Like all of us, students are more likely to succeed when they understand what is expected of them. Maintaining a steady rhythm in the classroom can help in this regard. Students who know that the same deliverables such as code, design, homework exercises, and completed reading assignments will be expected on a regular basis are less likely to become frustrated than students in courses where such demands seem to be ad-hoc.

A daily agenda can set expectations for what is expected both in and out of the classroom. A steady and predictable rhythm also affords more opportunities for students to explore and experiment with different approaches to studying and can lead to discussions about metacognition and its applicability to learning to write code. Figure 3 provides an example of a typical agenda that lays out the day's tasks along with the tasks expected outside of class.

Today's agenda	1. Quiz over assigned reading 2. Review homework 3. Code with me: Chapter 7, first three sections 4. Design homework solution
To be completed before next class	1. Finish homework (code, test, and debug) 2. Design homework solution

Figure 3. An Example of a Typical Agenda with Tasks Expected Outside of Class

3.14 Necessitate Design Documents before Coding

Through program evaluation, we noticed students in our capstone class were going straight to coding, spending little time on design prior to implementation. This made for poor code maintainability, reusability, and readability, and inferior overall software design.

A program design document helps the student connect the program requirements to design choices and then connect those design choices to implementation. We want our students to build the mentality of planning before coding. We also want them to build additional mental models of their solutions in addition to source code. The research results by Ramalingam, LaBelle, and Wiedenbeck (2004) show that the student's mental model of programming influences self-efficacy and that both the mental model and self-efficacy affect course performance.

After discovering design was a student weakness, we began to increasingly require design submissions for major projects prior to source code submissions. This evolution resulted in a design document template that students are now required to complete prior to every major project in CS I. A successful design document in our course will completely and unambiguously describe all of the following items: (1) program requirements, (2) inputs, (3) outputs, (4) a test plan, (5) a solution algorithm, and (6) a flowchart of the algorithm.

We introduce the design document well before the first major project. Prior to the first project, different aspects of this design document are modeled with in-class activities and portions of the design document are required for smaller assignments given outside of class. This activity helps the student think about the solution before they begin coding.

When we assign larger, one-week long projects in CS I, we require students to submit their design document two days after the project assignment is distributed. Students then have the remainder of the week (five days) to complete and test the actual source code. The design document not only forces the student to plan, but it also ensures that the student completely understands the problem before attempting to code a solution. We found that students with good design documents not only scored higher on their source code submissions but also performed better on exams (Terwilliger and Jenkins, 2017).

3.15 Offer Homework Feedback using Students' Code and Do the One-On-One Grading

Experience and research have shown that many students will make similar kinds of mistakes when learning how to program (Hristova et al., 2003). When the students' work is graded, common mistakes can be collected and one student's submission that contains one or more of these errors can be displayed to the entire class. To avoid embarrassing a student, the instructor should always remove the student name from the comment header before the code is displayed; some students just do not like being called out, even for praise. If time allows, all of the errors made by the students may be displayed to the entire class when graded homework is reviewed in order to ensure that all student questions can be answered.

Whenever possible, we as instructors, will grade students' program submissions with them one-on-one, especially for those students that need our help the most. Grading a student's program one-on-one can benefit both the instructor and the student. On the one hand, the instructor has a great opportunity to get to know the student better, to understand what help is most needed by the student, and to clarify the student's misconceptions. On the other hand, the student has a chance to get to know the instructor better, to understand the instructor's expectations, and to ask clarifying questions.

When a one-on-one session is not practical, a small group review session of three or four students can also accomplish the same objectives. A real-time, face-to-face code review session can provide students with "lots of pragmatic learning" (Guo, 2014). Specifically, students can ask clarifying questions and get prompt and understandable answers.

Timely communication and feedback is key to student learning effectiveness (Roth and Klein, 2012). Whenever possible, the student work should be graded and returned as quickly as possible in order to provide prompt feedback. Our goal is to grade and return student work within 72 hours of the assignment due date. There are many positive outcomes to providing students with prompt feedback. Students recognize that their instructor is engaged and wants them to succeed. Since the work should still be relatively fresh in the student's memory, it should be easier for the student to understand and relate to the instructor's feedback.

3.16 Provide Sample Runs for each Programming Assignment

Providing sample runs is useful for both instructors and students, especially in an introductory programming course. A sample run shows both the input and output interfaces. A simple Java programming assignment is provided in Figure 4.

JAVA_HA1.2: Three Means

Write a program (**ThreeMeans.java**) that prompts the user to enter two positive floating-point numbers and prints their arithmetic mean, geometric mean, and harmonic mean. When the user enters 4.2 and 2.7, the output of your program should look exactly like the following:

```
Enter two positive floating-point numbers: 4.2 2.7
The arithmetic mean is 3.45.
The geometric mean is 3.37.
The harmonic mean is 3.29.
```

Figure 4. An Example of a Simple Java Programming Assignment

For the instructor, the sample run makes it easier to grade the student's submission. Treating the sample run as part of the program specification, the instructor can use a black-box testing approach to run the student's submission and compare the result character-by-character with the sample run provided

in the program specification. Any deviation between the student's result and the sample run will result in a point deduction. For students, the sample run provided in the program specification gives them clear directions and also forces them to use specific techniques. For instance, with the prompt message in the sample assignment described above, they have to use *print* instead of *println*; otherwise, the two input numbers will be pushed to the next line. They also need to figure out how to format the output numbers so that each number has two digits after the decimal point. Finally, providing the sample run helps the students understand that programming can be viewed as a process of converting program specifications into working code. Furthermore, forcing the students to explicitly conform to the format of the sample run helps emphasize that defined interfaces cannot be modified by the programmer without the consent of all pertinent stakeholders.

3.17 Quiz Students Frequently

We have observed that student attendance and continuous active engagement are critical for success in introductory programming courses. Students seem to have more and more difficulty reading the assigned textbook material before coming to class. This is, no doubt, influenced by the media deluge that we experience on a daily basis. However, reading is invaluable in learning to program. The terminology and concepts are often new and can be quite abstract. A short quiz, limited to two or three minutes at the start of class, can serve as motivation for *actually reading* the assigned material so that the terms and concepts presented in the lecture have some degree of familiarity. Ideally, homework that covers the same concepts can assist in presenting the information a third time.

Students who admit to being lost with assigned homework have, at times, admitted that they have never read the material in the textbook. A daily quiz may provide the added incentive needed to begin the process on the right foot.

If a daily quiz is not desirable or feasible, another strategy to address this issue is giving unannounced or surprise quizzes. Class time is often precious, so the quizzes can take place in 5 to 10 minutes at the beginning or end of the class period. If class sizes are large or no grading assistance is available, students can still be assessed in an effective manner using multiple-choice questions. Figure 5 shows three sample questions to evaluate students in a CS I course that uses C++.

1. What is the output for the code snippet below?
(a) 35 (b) 36 (c) 35.77 (d) 35.78
 	
double b = 35.7777;
cout << setprecision(2) << fixed << b << endl;

2. What is the output for the code snippet below?
(a) 0 (b) 5 (c) 6 (d) 10
 	
num=0;
for (int i=5; i<=10; i++)
  num++;
cout << num;

3. What is the largest valid subscript for stuff in the following declaration: int stuff[8]; (a) 1 (b) 7 (c) 8 (d) 9

Figure 5. Sample Questions in a Daily Quiz

Even if the quizzes end up being a small percentage of the students' semester grade (e.g., 10%), they still should accomplish these two important goals: (1) improved course attendance and (2) students studying and reviewing course topics on a continuous basis. In fact, it may be possible to increase student final exam scores by giving pop quizzes and not even grading them (Khanna, 2015).

3.18 Read before Writing

Most children learn to read before they learn to write. Many learn to form letters as they are learning to read. Adults who are learning a second language must be able to read before they can possibly be expected to write. However, budding programmers are sometimes writing code before they really learn to read what experienced programmers have produced. Textbooks often have snippets of code or small simple programs, but rarely display a real-world example of non-trivial code.

Showing introductory students professionally-crafted code as an exemplar can have many benefits. If the instructor emphasizes the modularity of the code, students may be able to recognize the syntax for defining the scope of each module before they learn the language. The style of the code can make a good first impression that can be emulated. The proper use of operations or methods from the main program of a carefully chosen example can make the overall purpose of the program readable, even to beginners.

3.19 Steer Clear of Slide Decks

Students in introductory programming courses, as in most courses, tend to "zone out" during PowerPoint presentations. There have been many articles published that question the ability of such presentations to enhance learning (e.g., Szabo and Hastings, 2000; Penciner, 2013). While there is nothing wrong with presentation tools such as PowerPoint or Keynote in and of themselves, instructors should think carefully about the purpose of such tools. Penciner (2013) argues that while there are many reasons why lecturers use slide decks, there are only three reasons to consider when deciding if they would be appropriate for your next presentation:

1. Emphasis. A single word or phrase related to the concept conveyed.
2. Argumentation. Often using graphs or tables.
3. Multimedia learning. Pictures can often make verbal information memorable.

While using slides for emphasizing concepts or displaying pictures of internal memory structures may be useful, argumentation is rarely part of the typical computer course. It is even more difficult to imagine how relying on such slides for the bulk of the presentation can be justified in most modern classrooms where the instructors (and often the students) have computers loaded with the editors, compilers, and interpreters necessary to write, build, and execute code.

3.20 Try Multiple IDEs and Engage Students with Interactive Tools

An IDE typically consists of a source code editor, build automation tools, and a debugger, integrating the functionalities of editing, compiling, building, debugging, and

online help in one graphical user interface (Liang, 2015). Most IDEs provide features such as syntax highlighting and code completion, making it easier for programmers to write and debug their programs. There are many IDEs available for programmers to choose from, and most of them are free for personal and educational use. Each IDE has its own advantages and disadvantages. Thus, experimenting with multiple IDEs provides students the opportunity to navigate different interfaces, learn diverse features and focuses, and build confidence in using them.

Engaging students with different interactive tools can also improve their learning effectiveness. There are many interactive tools for students to use. One such tool is *CodeSkulptor*, which is an interactive, web-based Python programming environment that allows Python code to be run in a web browser (http://www.codeskulptor.org). With a web browser, the tool allows users to write, edit, run, debug, and test the code. It can animate program execution and provide cogent error messages that are helpful for debugging code.

Numerous IDEs provide an option to code with visual blocks, which allow novice programmers to focus on the code's semantics rather than its syntax (Bau et al., 2017). Scratch, for instance, is a free, block-based visual programming language. It was developed to "make it easy for everyone, of all ages, backgrounds, and interests, to program their own interactive stories, games, animations, and simulations, and share their creations with one another" (Resnick et al., 2009, p. 60).

3.21 Use Planned Schedules for Larger Projects

It is difficult to enumerate the tasks needed to complete a complex programming project. In a study of CS I classes, we evaluated how long students thought they would take to complete projects, how long they actually took to complete the projects, and how much of their time was spent on design prior to coding (Terwilliger and Jenkins, 2017). We examined how those factors influenced the amount of time during which students felt they were "stuck" and how it related to their overall performance.

Our objective was to have students begin the planning process immediately after the project is assigned. We wanted to get students thinking about the "big picture" of the project development life cycle. We wanted them to anticipate when they may get stuck so that it was during a time when resources such as tutoring and instructor office hours would be available. For a week-long project, students had to submit a spreadsheet that showed how much time they expected to be working on their project design, the actual coding, as well as the program documentation.

Once the project source code was turned in, the students submitted a second spreadsheet that detailed the actual hours invested, as well as other data such as what resources were used and how much time was spent "stuck." Perhaps not surprising, students did not plan enough time for their projects. They did get better at planning, however, as the semester progressed. We also observed that the better planners performed better on course exams.

3.22 Value Certification

There is much debate surrounding the nature of the relationship between academic degrees and industry certifications, especially as they apply to students and recent graduates of computing programs (Hitchcock, 2007). The value of such certifications varies with time, location, and experience. However, the right certification at the right time can sometimes give a candidate entrée into opportunities that are not available otherwise.

Many organizations (w3schools.com, Oracle, CompTIA, etc.) provide certifications that can be used to provide additional opportunities and act as motivation above and beyond letter grades. Oracle offers a Java Foundations Certified Junior Associate exam that may be appropriate for some introductory programming courses. Regardless of major, such certifications can help students find employment during school or after graduation. Helping students become aware of the available certifications, as well as having discussions about their value, is a great service.

3.23 Work with REPL Environments

Many languages, including LISP, Scheme, and Python, include a Read-Eval-Print Loop (REPL) shell or environment (Sandewall, 1978; Findler et al., 2002). Other languages, like Java, have recently added such support. These environments can be a valuable resource for beginning programmers. At the Python REPL prompt, students may enter entire programs, statements, or simple expressions. This environment is ideal for teaching students the difference between statements, which can also be a single line in a working Python script, and expressions which are not valid as a line in a program. It is also easy to demonstrate the difference between defining a function, which is stored at a particular memory address (which can be displayed by simply typing the name of the function), and the invocation of that function by using the name along with the parameter list. There are many uses for the REPL environment when teaching programming, and students can benefit by learning how to quickly jump back and forth between a program editor and the REPL prompt.

3.24 X-Out Student Frustration and Show that We Care

As computer programmers, we are accustomed to providing clear, concise, and precise instructions that can be unmistakably interpreted by a compiler or interpreter and then executed. However, as computing professors, we do not have the guarantee that our instructions will be so readily accepted by our human students. Humans make mistakes, they misunderstand, they are creative, they can lose focus, they are sometimes bored, or they get distracted by the birds. Beyond the basic error-prone nature of humans is the sheer frustration which can be overwhelming when students are learning to program. How the student deals with that frustration can be an indicator of how successful they become in an introductory programming course. Rodrigo and Baker (2009, p. 75) state, "researchers recognize frustration is potentially a mediator for student disengagement and eventually attrition." Affective factors (emotional factors impacting learning) are significant. Lishinksi (2016, p. 261) states, "affective factors (e.g., self-efficacy) and dispositional factors (e.g., personality traits) may be just as important as cognitive factors in learning to program."

We show our students that we care about them and their success in our course when we help them to manage some of the frustrations of learning to program. There are many ways

we can help our students deal with their frustrations in addition to enhanced instruction. We can be good listeners and talk to students with respect. We can be available to help students. When we offer constructive criticism, we should first offer a positive comment about their work. When students hit roadblocks, it is important to let them know they are not alone. Being nice to students does not mean we lower our standards; it just means we are treating them the way we would like to be treated. We have found that when students know we care about them, they seem to be more determined to do well in our classes. They know they have someone who will walk them through the hard parts while still showing them how to deal with the challenges of learning to program.

3.25 Yield Results beyond Teaching Syntax

When teaching programming classes, the instructor often explains a language construct and then makes up some elementary use of that construct. This is similar to showing someone how a hammer is used by driving a nail into a piece of wood. But why not use a more practical approach to show the student how hammers and nails are used to build something? "Programming can be viewed as a social practice structured by tacit 'rules of the game' rather than a formal exercise linking specifications to code" (Tenenberg et al., 2018, p. 66). Instructors need to emphasize using programming language constructs for problem solving and not just as syntactic entities. Lahtinen, Ala-Mutka, and Järvinen, (2005, p. 17) state that "the biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them."

For example, when teaching the modulo (remainder) operator, we normally just show how it works, try it out on the machine, and let the students practice it as well. This is not inherently wrong, but there are a number of applications where the modulo operator is a very useful tool. So, why not have smaller assignments where students can discover its use to extract digits, identify factors, print calendars, and implement circular queues while learning the syntax? It is unlikely that many students have practical experience in using the remainder of a division operation and this is a great opportunity to provide that.

When we teach students about variables of type character, why not create simple examples that help illustrate the fact that these variables are represented, internally, as numeric values? For example, students could be asked to print out the letters of the alphabet with a simple *for* loop:

```
for (char ch = 'A'; ch <= 'Z'; ch++)
    cout << ch << ":" << int(ch) << endl;
```

This gives them practice declaring and initializing characters, writing loops, and gives them a peek at how characters are represented in memory.

When teaching loop constructs, we typically use contrived scenarios to teach loop syntax and show the results of using those loops. But this is an opportunity to show students how the loop construct can be used as a problem solving tool. For example, we could ask our students to find the largest or smallest values in a list of numbers. This elementary action is an important sub-task in many powerful algorithms, and students need to be comfortable with its implementation by the end of their first programming course.

3.26 Zone in on Creativity

Coding is a very creative activity. There are often numerous ways to solve the same problem. Many people get joy out of coming up with a unique solution to a familiar problem, especially if that solution is efficient, elegant, or has some other quality that sets itself apart from the more common approaches.

While many students may associate creativity with graphics programming and attractive user interfaces, writing elegant algorithms can be a source of gratification for those who are encouraged to put their creative problem-solving skills to work. Therefore, it is important that we encourage students to think "outside-the-box" and apply their ingenuity when solving problems.

4. DISCUSSION

4.1 Summary of Content

We have presented a collection of 26 tips for teaching an introductory programming course. The list was produced by a team of four highly experienced faculty members who initially worked independently and then consolidated and refined their collection of tips to produce the alphabetical list in Table 1.

No.	Teaching Tip
1	Assign homework on debugging
2	Begin with an exciting application
3	Code early and often
4	Design and code exercises with published solutions
5	Emphasize code style and demonstrate conventional structures
6	Flip the classroom and let students take control
7	Gauge student concept mastery using exit tickets
8	Help students build computational thinking skills
9	Inject peer tutors into the classroom
10	Just go agile and team students up
11	Keep it simple
12	Learn from our peers and get student feedback
13	Maintain a steady rhythm
14	Necessitate design documents before coding
15	Offer homework feedback using students' code and do the one-on-one grading
16	Provide sample runs for each programming assignment
17	Quiz students frequently
18	Read before writing
19	Steer clear of slide decks
20	Try multiple IDEs and engage students with interactive tools
21	Use planned schedules for larger projects
22	Value certification
23	Work with REPL environments
24	X-out student frustration and show that we care
25	Yield results beyond teaching syntax
26	Zone in on creativity

Table 1. Twenty-Six Tips for Teaching Introductory Programming

4.2 Pedagogical Implications

The 26 tips for teaching introductory programming presented in this paper have important pedagogical implications. First, the collection of tips is based on years of personal experience, so all the tips have gone through the test-adjust-retest process and have, at least informally, been shown to work. Second, we have made every effort to ensure that the tips are clearly presented. For each tip, the *what*, *how*, and *why* questions are addressed. For instance, in the tip titled "Flip the Classroom and Let Students Take Control," we focused on the following three questions: (1) What is the flipped classroom model? (2) How do we apply the flipped classroom model in our teaching? (3) Why is the flipped classroom model effective in teaching and student learning? By clearly elaborating on each teaching tip, our peers can easily adopt a specific tip and apply it in their instruction. Third, these teaching tips can provide incentives for other introductory programming instructors to develop their own teaching tips. More well-elaborated and well-disseminated tips will certainly inspire more educators to join the effort and hopefully encourage the development of a virtuous cycle for improved teaching and learning effectiveness.

It is worth noting that our list of teaching tips, although comprehensive, is not meant to be exhaustive. We do not expect professors to apply all of our teaching tips in their classroom. Our goal in writing this paper is two-fold. First, we wanted to share our teaching tips so that our peers can apply them in their own classroom. Second, we wanted to inspire our peers to participate in the creation and sharing of their own teaching tips in an effort to improve overall teaching and student learning effectiveness.

4.3 Limitations and Future Research

This study has several limitations. First, the four-person group of instructors, all based at the same university, provides only an isolated glimpse into the variety of teaching strategies employed around the world. A team with a larger number of participants could surely generate more teaching tips, especially if participants are from different schools. An online questionnaire could be created so that more instructors could share their teaching experiences. Second, this paper is intended to provoke thought and encourage discourse. All of the tips presented in the paper are based on personal experience. Future research can design and implement more rigorous empirical studies to capture quantitative and/or qualitative data in an effort to validate premises that are untested in this paper. Third, teaching and learning are two sides of the same coin. In this paper, we focused on the teaching side. Future research can look more into the learning side. Research on tips for learning introductory programming developed by students is a promising area that could complement this study and shed some light on improving teaching and student learning effectiveness.

4.4 Concluding Remarks

Effective teaching of introductory programming is both important and challenging. The 26 tips presented in this paper should help introductory programming instructors improve their teaching effectiveness, thereby improving student learning outcomes. It is our hope that our peers can utilize some of the tips from this paper, apply them in the classroom, and immediately see the benefits for their students.

5. REFERENCES

Angelo, T. A. & Cross, K. P. (1993). *Classroom Assessment Techniques: A Handbook for College Teachers (2nd ed.)*. San Francisco, CA: Jossey-Bass.

Barnes, D. J. & Kölling, M. (2016). *Objects First with Java: A Practical Introduction Using BlueJ (6th ed.)*. Upper Saddle River, NJ: Pearson.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable Programming: Blocks and Beyond. *Communications of the ACM*, 60(6), 72-80.

Benander, A. C. & Benander, B. A. (2008). Student Monks – Teaching Recursion in an IS or CS Programming Course Using the Towers of Hanoi. *Journal of Information Systems Education*, 19(4), 455-467.

Chen, K. & Rea, A. (2018). Do Pair Programming Approaches Transcend Coding? Measuring Agile Attitudes in Diverse Information Systems Courses. *Journal of Information Systems Education*, 29(2), 53-64.

Crabtree, J. D., Nickels, D. W., & Parris, J. B. (2013). Clearing the Hurdles to Success in Teaching Computer Programming: Applying the Scientific Method to Improve Student Outcomes. *Academy of Business Research Journal*, 2, 45-56.

Dickson, P. E. (2011). Using Undergraduate Teaching Assistants in a Small College Environment. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 75-80). New York, NY: ACM.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, 12(2), 159-182.

Guo, P. (2014). Refining Students' Coding and Reviewing Skills. *Communications of the ACM*, 57(9), 10-11.

Hitchcock, L. (2007). Industry Certification and Academic Degrees: Complementary, or Poles Apart? *Proceedings of the 2007 ACM SIGMIS CPR Conference on Computer Personnel Research: The Global Information Technology Workforce* (pp. 95-100). New York, NY: ACM.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 153-156). New York, NY: ACM.

Khanna, M. M. (2015). Ungraded Pop Quizzes: Test-Enhanced Learning without All the Anxiety. *Teaching of Psychology*, 42(2), 174-178.

Kules, B. (2016). Computational Thinking Is Critical Thinking: Connecting to University Discourse, Goals, and Learning Outcomes. *Proceedings of the 79th ASIS&T Annual Meeting: Creating Knowledge, Enhancing Lives through Information & Technology* (pp. 1-6). Silver Springs, MD: American Society for Information Science.

Lahtinen, E., Ala-Mutka, K, & Järvinen, H.-M. (2005). A Study of the Difficulties of Novice Programmers. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 14-18). New York, NY: ACM.

Liang, Y. D. (2015). *Introduction to Java Programming, Comprehensive Version (10th ed.)*. Upper Saddle River, NJ: Pearson.

Linden, T. (2018). Scrum-Based Learning Environment: Fostering Self-Regulated Learning. *Journal of Information Systems Education*, 29(2), 65-74.

Lishinksi, A. (2016). Cognitive, Affective, and Dispositional Components of Learning Programming. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 261-262). New York, NY: ACM.

Marzano, R. J. (2012). Art and Science of Teaching: The Many Uses of Exit Slips. *Educational Leadership*, 70(2), 80-81.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A Review of the Literature from an Educational Perspective. *Computer Science Education*, 18(2), 67-92.

Mok, H. N. (2014). The Flipped Classroom. *Journal of Information Systems Education*, 25(1), 7-11.

Penciner, R. (2013). Does PowerPoint Enhance Learning? *Canadian Journal of Emergency Medicine*, 15(2), 109-112.

Radermacher, A. D. & Walia, G. S. (2011). Investigating the Effective Implementation of Pair Programming: An Empirical Investigation. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 655-660). New York, NY: ACM.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-Efficacy and Mental Models in Learning to Program. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 171-175). New York, NY: ACM.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Siverman, B., & Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, 52(11), 60-67.

Rodrigo, M. M. T. & Baker, R. S. J. D. (2009). Course-Grained Detection of Student Frustration in an Introductory Programming Course. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (pp. 75-79). New York, NY: ACM.

Roth, Y. & Klein, D. (2012). Effective Teaching Elements in Online Adult Learning. *Issues in Information Systems*, 13(2), 155-163.

Sandewall, E. (1978). Programming in an Interactive Environment: The "Lisp" Experience. *ACM Computing Surveys,* 10(1), 35-71.

Sheard, J., Simon, Butler, M., Falkner, K., Morgan, M., & Weerasinghe, A. (2017). Strategies for Maintaining Academic Integrity in First-Year Computing Courses. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 244-249). New York, NY: ACM.

Shein, E. (2014). Should Everybody Learn to Code? *Communications of the ACM*, 57(2), 16-18.

Sommerville, I. (2016). *Software Engineering (10th ed.)*. Upper Saddle River, NJ: Pearson.

Szabo, A. & Hastings, N. (2000). Using IT in the Undergraduate Classroom: Should We Replace the Blackboard with PowerPoint? *Computers & Education*, 35(3), 175-187.

Tan, P.-H., Ting, C.-Y., & Ling, S.-W. (2009). Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception. *Proceedings of the 2009 International Conference on Computer Technology and Development* (pp. 42-46). Piscataway, NJ: IEEE.

Tenenberg, J., Roth, W.-M., Chinn, D., Jornet, A., Socha, D., & Walter, S. (2018). More than the Code: Learning Rules of Rejection in Writing Programs. *Communications of the ACM*, 61(5), 66-71.

Terwilliger, M. & Jenkins, J. T. (2017). Exploring the Relationship between Planning and Problem Solving with CS1 Students. *Proceedings of the 59th Annual ACM Mid-Southeast Conference* (p. 76). New York, NY: ACM.

Trujillo, G., Aguinaldo, P. G., Anderson, C., Bustamante, J., Gelsinger, D. R., Pastor, M. J., Wright, J., Márquez-Magaña, L., & Riggs, B. (2015). Near-Peer STEM Mentoring Offers Unexpected Benefits for Mentors from Traditionally Underrepresented Backgrounds. *Perspectives on Undergraduate Research and Mentoring*, 4(1), 1-13.

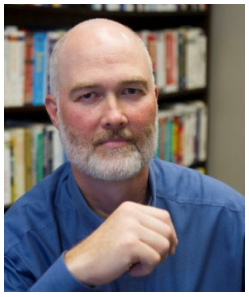
Westfall, R. (2001). Technical Opinion: Hello, World Considered Harmful. *Communications of the ACM*, 44(10), 129-130.

Zhang, X., Hu, T., Dai, H., Li, X. (2010). Software Development Methodologies, Trends and Implications: A Testing Centric View. *Information Technology Journal*, 9(8), 1747-1753.

Zhang, X., Zhang, C., Stafford, T. F., & Zhang, P. (2013). Teaching Introductory Programming to IS Students: The Impact of Teaching Approaches on Learning Performance. *Journal of Information Systems Education*, 24(2), 147-155.

AUTHOR BIOGRAPHIES

Xihui "Paul" Zhang is a professor of computer information systems in the College of Business at the University of North Alabama. He received his B.S. and M.S. degrees in earth sciences from Nanjing University (1993, 1996), Nanjing, China, and his M.S. and Ph.D. degrees in management information systems from the University of Memphis (2004, 2009), Memphis, TN. His work has been published in the *Journal of Strategic Information Systems*, *Information & Management*, *Journal of Database Management*, *Journal of Organizational and End User Computing*, *Journal of Computer Information Systems*, *Journal of Information Systems Education*, and *Journal of Information Technology Management*, among others. He serves as the managing editor of *The Data Base for Advances in Information Systems*. He also serves on the editorial review board for several academic journals, including the *Journal of Computer Information Systems*, *Journal of Information Systems Education*, and *Journal of Information Technology Management*.



John D. Crabtree is a professor of computer science and information systems in the College of Business at the University of North Alabama where he has taught courses in e-commerce, database management, computer programming, software engineering, enterprise architecture, data structures, and algorithms. He earned his Ph.D. in computer science, an M.S. and B.S in math/computer science, and a B.S. in geophysical engineering, all from the Colorado School of Mines in Golden, Colorado. Prior to his career in academia, he was a software development consultant in the Denver area where he developed systems in e-commerce, telecommunications, science, defense, GIS, manufacturing, oil and gas exploration, and insurance. His research interests include software engineering, GIS, big data, graphing algorithms, and cheminformatics.

Mark G. Terwilliger is an associate professor of computer science in the College of Business at the University of North Alabama. He earned a Ph.D. in computer science from Western Michigan University in 2006. His dissertation was on the topic of localization in wireless sensor networks. He earned a Master's in computer science at Michigan State University where his focus was on artificial intelligence and expert systems. He was previously a professor in computer science and mathematics at Lake Superior State University, where he also served as the chairman of the computer science and mathematics department. His research publications have been in the areas of wireless sensor networks, evolutionary computing, parallel algorithms, and computer science education. He currently serves on the program committees for the *International Conference on Agents and Artificial Intelligence* (ICAART) and the *International Conference on Intelligent Systems and Applications* (INTELLI).

Janet T. Jenkins is an associate professor of computer science at the University of North Alabama. She has taught computer science at the college level for over 20 years. She earned her Ph.D. and Master's in computer science at the University of Alabama (2008, 1999) with her research focus on software engineering and non-functional requirements, component-based software engineering, and the Common Object Request Broker Architecture. She earned her B.S. in secondary education with areas of mathematics and computer science in 1997. Her current pedagogical research interests are in the areas of computational thinking, computer science education, and mathematics education. These efforts have been funded by Math Science Partnership and are now funded by the National Science Foundation. Funding supports the work of training math teachers to apply an instructional model designed to explicitly teach generalization and abstraction by using computer programming to explore mathematical concepts.