An Experience Report of Our Teaching Visual BASIC using a Problem-Oriented Approach

J. Philip East & Stephen Hughes
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50614-0507
east@cs.uni.edu & stephen.hughes@uni.edu

Abstract

In considering the role of introductory programming as a service course to a range of disciplines, it is important to consider how new material is motivated and communicated to the students. Students who do not wish to become professional programmers probably care very little for syntax structures and other nuances of programming languages and are driven instead by a singular focus: "How can programming help me to solve this problem?" Considering this student perspective was the basis for redesigning two sections of Visual BASIC programming in the fall of 2012.

This paper addresses our thinking at the start of the class preparation and provides rationales for the various principles that guided our efforts. It also describes the course structure that reflects a shift from a language-feature to a problem-centric approach. Finally, it discusses our reactions, insights, and recommendations for future explorations of teaching introductory programming as a service course.

1 Background

Some faculty members keep to themselves about their teaching while some are relatively communicative; particularly about course organization—content, its sequencing, assignments, and assessment. We describe here an experience that began with a chance discussion and resulted in a substantial on-going consideration of student learning and teaching. When referring to "student learning" we include what we perceived students to be learning; the learning they demonstrated; and our notions of their thoughts when encountering class activity and assignments. We include various ideas and activities under the term "teaching": general approach in the course; desired student outcomes; homework assignments; quizzes and exams; and planning and performing actual class activity.

1.1 A Course Focused on Solving Problems

The chance discussion noted above was one in which Stephen indicated his desire to teach programming by walking into class on the first day with a list of 100 problems and announcing that by the end of the semester students should be able to solve any of them. Students would approach the class by categorizing the problems according to difficulty and learning the necessary programming skills "just-in-time". This was compatible with Philip's assertion that students are already *conceptually* familiar with the ideas involved with if statements and loops. Students would have some basis for differentiating the problems. We found some time to talk about these ideas and ultimately requested (and were assigned) to each teach a section of Visual BASIC (VB) in the fall semester of 2012.

Our experience with introductory programming classes and students echo the conclusions of McCraken et al [5] and Lister [4]; that students don't program very well after the first course. This would seem to be even more the case in a non-majors class. We suspect that these outcomes are related to an instructional emphasis on programming language features. We agreed that we (and, we believe, most other computing instructors) normally teach programming from the standpoint of language functionality, constructs, and syntax. That is, we organize our courses by ordering the progression from data types (and I/O) to if statements to for loops and while loops to procedures and so on. When we plan class, lab, and homework activity, we tend to select problems based on how well they demonstrate the particular language feature, often neglecting the relevance of the problem to the students' context.

We were not necessarily thinking that a language-feature focus was bad or inappropriate. We did note, however, that while many of us claim that our programming courses teach problem solving, students believe the goal is to learn a programming language. While teaching programming, we seldom mention problem solving in class and tend to focus on programming code—what it does, and how to make it achieve a stated objective. Rather than attempting to master the intricacies of a programming language, a more reasonable goal might be to expose programming as a tool that students can apply to the challenges they face in their native discipline.

We believe that if problem solving were a principal goal of a programming course it should be addressed directly rather than assuming students would pick it up. Our view of learning includes several related ideas compatible with accepted views of the research presented by Bransford, Brown & Cocking [1]:

- We learn from experience. As people go through life their brains record relationships between situations, actions, and outcomes. Positive outcomes will likely yield similar actions upon presentation with a similar situation and negative outcomes will probably yield different actions. The brain automatically processes past experience with whatever is happening at present.
- The brain generalizes from experience. Whatever pattern from memory that the brain settles upon will be used as a match for the current situation. The memory pattern will be affected by the current situation—the pattern becomes more generalized. In this sense transfer of learning is always occurring.
- Explication of process and meta-knowledge aids learning. Not only should students be able to solve problems they should understand how they solve the problems. This will allow them to be better able to solve novel problems in the future rather than just apply learned strategies.

Therefore, having students explicitly address problem solving should be a principal goal in our problem-centric programming course.

2 The Experience

We began planning the course during the summer of 2012. We met as our separate schedules allowed, perhaps about four weeks of time overall. When classes began we had relatively detailed plans for about the first four weeks and a general structure for class activity. Philip missed half of the third and all of the fourth week due to conference attendance (and Stephen covered both sections).

2.1 Planning and Guiding Principles

Initial discussions in the summer of 2012 centered on understanding the meaning and implications of focusing on problems rather than language features. We also worked on generating a list of distinct problems we felt would illustrate the kinds and complexity of problems that might be addressed by end-user programmers[1]. We felt the list of problems should apply to a variety of disciplines and activities. The discussions allowed us to get to know each other as we examined our beliefs about learning and teaching, programming,

[1] We adopted the view that in teaching non-majors we were attempting to develop end-user programmers who would know how to and feel comfortable writing programs to accomplish tasks they encountered during their daily lives.

teaching programming, skill versus knowledge with respect to programming, etc. Often discussions were about a particular idea or question that occurred to one of us and seemed not to have no common focus or contribute directly to developing the course plans. We reach a number of conclusions (or made assumptions) that would guide our thinking and approach to the course.

2.1.1 Students as Problem Solvers

We believe students already understand most of the underlying concepts of programming. In particular, the concepts of conditional execution, repetition, and modularization are familiar to them. They know how to make decisions depending upon some characteristic of the current situation, e.g., take an umbrella if rain is forecast. Similarly, they know about and use repetition though perhaps they rarely use the terms “repeat” or “loop”, e.g., shoot until the target is hit or ammunition is depleted. And, they understand abstraction via modularization. We think the difficulty students have with programming ifs, loops, and modules is based primarily in an unfamiliarity with representing problems abstractly using variables to generalize their solution. Additionally, students understand the need for precision when giving directions, or developing an algorithm, but are not fluent in doing so (probably) because such activity is typically verbal and interactive allowing for real-time correction.

2.1.2 Implications of Beliefs about Learning

Our beliefs about teaching in light of what is known about learning can probably be summed up with two statements, "**Have students do what you want them to be able to do**" and "**Learning is experiential**". Embedded in these statements are assumptions about knowledge versus skill, the importance of practice, and teaching for transfer.

We believe knowledge is important only because it is necessary for accomplishing some desired task. Being able to reliably accomplish particular tasks is the goal of instruction, not gaining knowledge necessary for the task. Knowing (being able to recite) the physics of throwing a baseball and the different grips and bodily motions for various pitches says nothing about one's ability to throw a curve ball in or near the strike zone. The knowledge is useful (though it need not be explicit) but skill requires deliberate practice. We often say we want our students to understand something but the understanding is always necessary for an actual capability (at least there should always be such a capability).

Practice is necessary for skill development. In our class, the desired skill is to be able to interpret a problem statement, model a solution, and develop a correctly functioning program meeting the needs of the problem. The experience involves the programming language, but probably more importantly it involves designing the algorithm for the program. Our goal is not the development of "expert" programmers so the 10,000 hours of deliberate practice noted by Ericsson & Charness [3] is not required. Substantial practice is required, however, if the students are to become fluent enough to continue programming after the course. We are concerned primarily with the solution-design skill and believe the necessary language familiarity or skill will be picked up in the process (note the reversal here). With enough practice the brain will generalize experience in a useful way.

Teaching for transfer is mostly a matter of practice but the problems need to be selected so that generalization can also occur, thus allowing for the solution of a variety of kinds of problems. In particular, end-user programmers will typically have a problem domain and some problems from those domains need to be included in the practice set. Part of the needed variety of problems can be achieved by selecting problems with different kinds of data and manipulation. However, some attention to the disciplinary domains of the students is also needed. Our approach to doing so would be to have the students identify problems similar to those in discussed in class that were in their own disciplines. Asking students to develop their own problems (and solutions) not only transfers the relevant skill into their native domain, it also forces them to explicitly think about the characteristics of problems where this skill is and is not applicable.

2.1.3 Problems and Programming Instruction

Obviously our plan was to focus on problems and developing solutions for them rather than on language characteristics. However, one must consider the possible role/impact of the prerequisite structure in a programming language. A discussion of prior work on within-a-language prerequisites, East [2], reaffirmed the conclusion that the tree showing prerequisites is broad and shallow with the concepts of sequential execution of statements, data literals and variables, and arithmetic and relational expressions at the top and nearly everything else in the next layer.

The initial plan was to work with students to examine and classify problems to gain insight into problem characteristics and complexity and solution approaches. This would be done repeatedly with an accompanying problem solution development. We generated a number of problems and examined them to determine how they might be categorized. Our discussion resulted in a classification very similar to what one might use if planning according to language features: I/O based problems requiring input, processing and output (IPO); problems involving conditional execution of some action(s); and problems in which repeated execution of some action(s). Each category would contain problems of greater or lesser complexity and complexity would be increased when elements of multiple categories occurred in a problem. While there is much similarity with the language-feature orientation there are also differences. One could include modules in IPO or conditional-execution problems rather than waiting for iteration (as is typical). Additionally, in GUI-based problems repetition can occur through user action rather than a language feature.

During our discussions of problems (and later while we were teaching) we would occasionally catch each other appearing to think in terms of language features rather than from a problem-centric perspective. It is difficult to describe precisely what that means but we want readers to know that it is likely to occur and that anyone using the problem-centric approach should take care to recognize it.

2.1.4 Gaps in Knowledge

In discussing the formulation of problem statements, we realized that students would encounter two primary sources of difficulty in solving the problem. In some cases there would be a lack of domain knowledge and in other cases there would be a lack of knowledge concerning how to accomplish a task using programming statements. We considered it valuable for students to experience both of these difficulties. Therefore, problems were often developed to not be completely self-contained. For example, student might be confronted with the problem, “Is this person eligible to vote in the state of Iowa?” with the rules for eligibility deliberately omitted from the problem statement.

2.1.5 Instructional Focus

The principal difference between the language-centric and problem-centric approaches shows up in class discussion. In both cases something is shown or developed and then discussed. In a language-centric approach that something is code. In the problem-centric approach that something is a solution plan. In both cases an instructor is likely to refer to prior work which will be previously seen code or previously seen problem solutions.

It is this focus and the discussion engendered by it in the class that we believe is critical if students are to perceive that the course is about solving their problems rather than the programming language used. New topics were to be introduced in the context of a concrete problem. After the initial presentation of the problem, the discussion would start with a question on the order of “What is different about this?”

2.2 Instructional Activity

Our plans gelled into eight units that would be similar in structure. The final assignment was an individually chosen, discipline-specific (or of personal interest) problem solution. Each unit would have a specific topic (mostly) based in problem categories, a homework assignment, and a unit quiz. Units would last from one to three weeks. Homework assignments involved completing a number of exercises that were similar in nature but diverse in the problem subject matter. Quizzes would generally be in-class programming tasks involving a single or a few small problems. Grades would be based on a combination of homework submissions, quiz scores, and project completion. We planned for homework assignments to require three to six hours of time each week. In some cases students were given starting points to use in their work and in other cases they were expected to produce everything that was needed by their programs. Normally, in-class discussion and demonstration would address all knowledge needed by the students but the problem-centric focus made the point that often there would be a need for additional domain or programming language knowledge. Demonstrations of seeking both kinds of information were included. Elements of VB would be introduced as they were necessary for solving the problems being addressed.

2.2.1 Introduction to the Course and Problem Solving

The first unit was designed to underscore students' existing ability to problem solve and to have them begin to focus on the process of problem solving. In an early activity, students were given a deck of playing cards with one card missing. Their objective is to identify the missing card. Working in groups, students were asked to describe *how* to solve this problem. The activity forces students to think algorithmically and to provide a reusable, detailed set of instructions. It is similar to the classic "how to make a peanut butter sandwich" exercise, but it is a more constrained, computational problem. In sharing their solutions with the class, student language was generalized into common computing concepts:

"First", "next", "then"	Sequence
"Count"	Arithmetic
"For Each Card"	Repetition
"Go Through"	Repetition
"Form Piles"	Collections; Storage
"If it is a…"	Conditional
"Is it a"	Comparison
"Sort"	High-level, abstracted task

The kinds of manipulations available in the programming language have a considerable influence in the steps in the algorithmic solution. As students become more familiar with programming, it will shape the way they approach new problems. The card task exposed students to many of these ideas, but it was important to formalize these fundamental concepts of computing as a basis for analyzing additional problems. These included: action statements (input, output, calculation, and storage), sequencing actions, executing actions conditionally, repeating actions, and grouping actions as modules. A number of problems would be solved in terms of these computing concepts during class discussion. An assignment would ask students to provide solution plans for similar problems and to identify similar problems in their home disciplines or of personal interest to them.

2.2.2 Representing Solutions in VB

The next step was for students to become familiar with the VB programming environment and converting problem solutions to code. At this point the goal was that students would be able to produce forms by copying examples and, with that experience, to develop original forms that could be used as the basis for thinking about problems and their programming solutions. In terms of problem solving, this unit placed emphasis on interpreting problem statements to understand the required inputs and desired outputs.

Once that can be clearly articulated, the student is ready to design a Visual Basic form that can support the problem. Thus, it makes sense to specifically address form production (if VB is the language being used). Homework included the development of seven forms: three to reconstruct from an image, three to develop from scratch, and one to develop from scratch and to solve with a program.

2.2.3 Variables and Expressions

We believe that variables and expressions, particularly Boolean expressions, are critical to programming problem solution and are mostly unfamiliar to students. Therefore, we planned a separate treatment of these topics before discussing conditional execution or repetition. The assignment involved evaluating and generating Boolean expressions, recognizing and fixing common errors in Boolean expressions, and identifying the necessary variables and creating Boolean expression to ask a question in a program, e.g., "Is overtime pay required?"

Of note here is phrasing of the central idea. We want to be able to **ask questions of the data**; at this point we were not concerned with how to interpret the answer. Thinking in those terms allows an approach different from "designing if statements".

2.2.4 Conditional Execution Problems (IFs)

By now students will have had experience with both problem solving and translating solutions to program code though experience with the latter will be minimal. However, the paper-based experience in the previous unit on variables and expressions would be expected to lower the overall difficulty of this unit. Class activity would demonstrate problem understanding and algorithm development followed by translation of the solution to VB code. The homework assignment would ask students to solve six generally stated problems requiring several conditional tests, one more complex and specified problem, and the identification of and solution to a problem of interest to the student.

2.2.5 Problems Requiring Repetition

In considering how to approach repetition we made note of two ideas. First, in a GUI interface, iteration can occur through repetitive user action. Class activity demonstrated the idea of user-initiated repetition and students were challenged to think about how to automate this prior to the introduction of VB code demonstrating repetition language constructs. Second, there are a number of different types of problems that require repetition. It would be helpful for transfer if some were identified and examples were provided that could be used as code templates for later development by students. The problem types were: data search, simulation, and data or report generation. Students were asked to solve problems of each type (seven programs in total) and to identify problems of each type that were from their home disciplines or of interest to them. The utilization of modules could be introduced at this time assuming a careful selection of examples and assigned problems.

2.2.6 Problems Using or Generating File Data

An additional category of problem types includes problems that use or produce/modify file data. They typically allow for the combination of many if not all of the problem types and language features already addressed as well as requiring some new language features. They can also be used to show the benefit of code modules. Class activity would again demonstrate problem examination and solving and solution translation to code. The assignment asked students to solve five problems.

2.2.7 Extending Problem Solutions and Modularization

The last unit having an assignment common to all students involved extending a sample program in two of a number of allowable ways. Modularization was required. In our case, we had built a dice rolling capability and used it to play a game. Students were expected to modularize that code and use it to add two new games to the program. Again, this assignment would allow the use of most if not all previous problem aspects and language features.

2.2.8 Individual Student Project

The last unit of the semester was one in which students were to identify a problem in their own discipline or a problem that they were interested in programming and to write a program to solve the problem, accomplish the identified task, or provide the desired capability/experience. The project required the use of much of what had been introduced in the course: a well-designed interface with a variety of elements, processing or manipulating data, some sort of internal data structure, file input or output, modules, and interacting forms.

2.3 Reactions and Conclusions

The course was similar to the first offering of any course. It was beset with small problems. Additionally, we realized that, similar to earlier experiences about student capabilities after a programming course [4,5], our students were probably not particularly adept at programming. However, we didn't feel we had utterly failed the students and our beliefs about the problem-centric approach remained basically the same as when we started.

Grading was a problem, particularly when a considerable amount of time had to be expended in planning instructional activity and assignments. The next version of the course would need to require less grading without substantially reducing the amount of practice that students would get.

Related to grading was the realization that we were probably seeking "mastery" on the part of students. Reflection and discussion led to the conclusion that the mastery probably related to the essential programming capabilities—variables and expressions, actions and their sequencing, conditional execution, and repetition—and to possessing capability with VB forms and programming assignment submission. Further, this mastery should be a requirement for a passing grade, not as part of the determination of which passing grade was merited.

While there were some problems with what we did there were also some (sometimes serendipitous) successes. We felt good about separating the variables and expressions topic and the assignment we produced for that unit. And, we think formulating the general notion of conditional execution (and loop termination) as asking questions of the data helps free us from the need to use terminology related to language features and is a good way to communicate problem solving activity to/with students. We reused/revisited some of the early assignment problems in later assignments or in class demonstrations. That is good since it cuts down on work for us and on the cognitive load imposed on students. However, it may also reduce the problem diversity student encounter and lessen transfer.

Finally, we cannot overstate our enjoyment of working together on an examination of our practice and understanding of teaching and learning. At the heart of many of these conversations was a focus on student thinking. The retrospective of the days' lesson almost always included a discussion of student misconceptions or gaps that still needed to be bridged to complete the students' understanding. This would often lead to the articulation of new problems that could be explored in subsequent lessons. The discussions have been marvelous and, we believe, beneficial to each of us in terms of both our understanding and our practice.

3 The Next Semester

Only one section of VB was to be offered in the next (spring) semester. After considerable reflection it was decided that Stephen would continue teaching VB. Our discussions continued though to a lesser extent. While the focus remained the same it was redirected a bit and the implementation was refined.

While we wish to emphasize problem-solving skills, a basic understanding of the language features cannot be overlooked. In the spring term, students will be required to "master" the basics of VB interface development and the essential programming concepts of variables and expressions, actions and their sequencing, conditional execution, and repetition. Students must pass a fluency-level quiz in each of these topics to be eligible to pass the class. Unlike traditional quizzes, these are not designed to determine a student's grade based on how much is known at the time of evaluation. Instead, students are allowed to repeat each quiz until a core competence in the topic has been demonstrated. These capabilities are deemed to be necessary but not sufficient for problem solving with computer program solutions. The student's grade is ultimately determined by homework and projects that are rooted in the problem solving metaphor.

Another change is that the notion about sources of difficulty in programming is being used more explicitly and has been refined a bit. We now perceive three possible considerations when asking, "Why can't you solve this problem?" It might be because you don't know how to accomplish the desired task in the programming language being used. It might be because you don't have the necessary domain knowledge. (And, newly added) It might be because you have no idea how to proceed in attempting to develop a solution. As programming instructors we often focus on (or stop at) the first question. If we remember to consider the second source of difficulty we will be able to help students overcome some problems that we hadn't considered. The third difficulty is one we seldom think of. It is probably actually quite common and unrecognized. Being able to overcome it is also likely at the center of whatever allows students to be good problem solvers and programmers. By at least being aware of that possible source of student difficulty, we are likely to be able to better help students learn to solve problems.

Another simple question is being used in order to engage students and enhance problem solving, "What would make this program better?" Once an in-class solution has been developed, we can ask this question to encourage students to develop a more general solution, or to add functionality. These student-driven modifications provide additional practice at a minimum. However, more than likely, the suggestions will extend beyond what they can currently accomplish and will motivate the need to learn additional programming skills.

4 Recommendations

First and foremost, we heartily recommend that instructors talk with others about their teaching goals and practices and the rationales for them. It has worked extremely well for us. There are, however, some caveats. The initial relationship must be a comfortable and trusting one. It may not be a good idea to expose your doubts or failings to your department head. Additionally we each have some background in education—Philip's initial training was as a secondary mathematics teacher and his graduate work included both computer science and education. Both have previously participated in discussions of teaching and learning. You may not have as much fun as we are having but we feel such discussions cannot help but help.

We recommend focusing on problem solving when teaching programming, particularly in non-major or service courses. While majors may need or want (or at least accept) much of the language-centric material, it is not useful for non-majors or end-user programmers. If you choose to try the approach, change small elements with each offering rather than going whole hog. If you do make a major revision try very hard to have much of the planning done (topics planned, assignments produced, etc.) before the course begins.

We also recommend you consider some of the actions noted above that we thought useful whether you take a problem-centric approach or not. Treating variables and expressions separately is a good idea. Considering and perhaps separating course content as we did with respect to mastery (minimal skill required for passing) and capability that should determine level of a passing grade seems useful. Similarly separating concerns seems useful with respect to student practice/homework and grading. Decide on homework based on what seems necessary for skill development not on what can reasonably be graded.

Picking assignment and class demonstration problems is important. The diversity of problems chosen will critically impact how, if at all, students will use/apply programming in the future. The problems should include a variety of data: numeric, textual, image/audio, etc.; user and file I/O; simple and structured data; etc. The problems should illustrate various disciplines or genre—not all games or CS topics (e.g., searching and sorting). Additionally, problems that might be reused or revisited should be considered and extending or modifying solutions is reasonable.

Finally, we believe that focusing on student learning can be extremely useful. Thinking about the possible sources of difficulty when programming is a tool that can aid that effort. Trying to imagine what the students are thinking as they see our demonstrations, make errors, or take assignments in an unexpected direction will also be helpful. Generally, our own learning experience has led us to focus our teaching efforts on presenting the content. Unfortunately, identifying and overcoming student learning difficulties cannot be aided by a focus on the content or lead to teaching that will.

You are welcome to contact us if you have questions or would like someone to talk with.

References

[1] John D. Bransford, Ann L. Brown, and Rodney R. Cocking (Eds.). 1999. *How People Learn: Brain, Mind, Experience, and School*. National Academy Press.

[2] J. Philip East. 1987. Analyzing the prerequisite structure of programming content. Presentation at 26th Annual University of Northern Iowa Fall Mathematics Conference (September, 1987) Cedar Falls, Iowa.

[3] K. Anders Ericsson and Neil Charness. Expert performance: Its structure and acquisition. *American Pyschologist* 49, 8 (August 1994), 725-747.

[4] Raymond Lister. 2011. Ten years after the McCracken Working Group. *ACM Inroads* 2, 4 (December 2011), 18-19.

[5] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz.

2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.* 33, 4 (December 2001), 125-180.