Teaching Programming as Application Development from the Ground Up

David R. Mudgett and Steven R. Haynes
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA, USA

Abstract— **This paper explores the proposition that teaching programming for application development differs from established methods in computer science, engineering, and MIS, and requires a re-framing of pedagogical models. In addition to programming fundamentals, algorithms, and data structures, application development also requires understanding the foundations of human-computer interaction and the nature and economics of information in different application domains. Tradeoffs to make this possible within a four-year undergraduate degree are explored using a revised version of Bloom's taxonomy of educational objectives to compare learning objectives for an applications developer curriculum with those for a traditional computer science curriculum. The paper also reports our experiences over the last five years with a new application design and development curriculum in the College of Information Sciences and Technology (IST) at Penn State University. Faculty and student backgrounds and interests range from business to social sciences to humanities to engineering and computer science. Students entering the major have diverse educational backgrounds different from students typically entering physical science, engineering, or computer science programs. We elaborate on these differences and discuss this new curriculum developed to focus more directly on the requirements of application development where they differ from those in numerical or systems-oriented programming.**

Keywords—programming; application development; curriculum; pedagogical models

I. INTRODUCTION

Over the last 20 years, there has been explosive growth in computer and network accessibility, with resultant demand for software to exploit this access. Today's user base has a very broad demographic profile. Software for the vast majority of these users also has a very different profile than that for traditional technical and business users. University training in software development has evolved but not significantly changed in this period of evolution. The traditional university route of developer training has been largely computer and information science/engineering departments, which are typically part of natural science or engineering colleges. There have always been other routes such as management information systems (MIS), and there has been some movement towards independent information technology colleges.

There is a significant base of prior research related to teaching introductory and intermediate programming in a range of disciplines from computer science and engineering. These focus on algorithms, numerical methods, and low-level systems programming; to management information systems, largely concerned with development of transaction-based systems and reporting; and end-user programming, which focused on topics such as spreadsheets as development platforms, and the usability of visual languages. These pedagogical frameworks are compared and contrasted in light of new models of learning application development, both within and outside of academia.

The paper also explores differences in the type of academic background appropriate for application developers. The prerequisites and technical course requirements for a typical computer science program can act as a significant barrier to entry, and many practicing, successful developers have educations from outside this established norm. Over the last 20 years, computer science and engineering have become increasingly complex, while in parallel there has been explosive growth in the demand for application developers. Computing now cuts across virtually all domains of human activity and is no longer the exclusive province of traditional software consumers such as science, engineering, and business.

One reaction to this growth has been rekindled interest in teaching programming and application development outside of traditional computer science programs. For example, since 2011 there is also an emerging set of over 60 so-called "code academies" which according to a recent study [1-2] trained about 16,000 graduates in 2015. This is almost a third the number of annual computer science graduates from accredited U.S. computer science departments. The training model in these bootcamps is very different, and concern has been expressed in some quarters that the training lacks sufficient depth in critical computer science and analytical problem-solving areas. Code academies are discussed later in the paper.

For university-based educators of application developers interested in competing for the market of would-be application developers, an essential question is: "What is the critical but minimal subset of computer science, science, mathematics, technical, and problem-solving skills required to be an effective developer in the current professional landscape and how can these skills be taught to a larger and more diverse student audience?" In this paper, this question is addressed by considering a number of sources along with our own experiences launching a new Application Design and Development program.

II. Historical Trends in Programming Education Pedagogy

Up until the 1990s, computer programming and development was largely the domain of science, engineering, and business. Science and engineering developers were generally trained in science and engineering colleges and used languages that required this training such as FORTRAN, C, Algol, and others. Business developers came from a wider variety of backgrounds, were often trained in business colleges or specialized data processing schools, and typically used COBOL, PL/1, or specialized business and report-generating languages. But since the 1990s desktop computing, the world-wide web, and increasing reliance on graphic user interfaces and complex APIs has led to significantly increased software complexity with consequent demands on the preparation of software developers.

A. Programming in Computer Science and Engineering

Typical computer/information science or engineering students often have strong high school science and math backgrounds, frequently have completed Advanced Placement (AP) courses in math, computer science, and other physical science areas, and are well steeped in the scientific method upon entering college. The typical CS Bachelor's degree core [3]-[5] requires at least two or three full courses in calculus for science/engineering, including vector calculus, matrix analysis, and often differential equations. Programs may also require one or two terms of calculus-based physics (mechanics, heat, and electricity/magnetism), one or two terms of basic chemistry, and sometimes core engineering courses like analog electric circuit analysis. There is usually a two-semester basic/intermediate programming sequence freshman year, typically followed by courses in computer organization and system programming, digital logic circuits, object-oriented programming (often with graphic and event-driven programming), operating systems, data structures and algorithms, discrete mathematics, computer architecture, theory of programming languages, theory of computation, and software engineering. Typical languages are C/C++, Java, Python, functional and/or logic languages like ML and Prolog, and some type of assembly language for system programming. For those aspiring to be a computer scientist, this is a fairly minimal background set to prepare for a highly technical development career or graduate school.

In an attempt to improve learning outcomes, there has been some movement towards active models of learning, such as problem-based-learning (PBL) [6, 7]. Concerned about deficiencies reported in CS students [8, 9] and recent CS graduates [10, 11], Radermacher and Walia [12] identified the primary deficiencies as communication and team skills. PBL has also being studied to improve issues with CS student team skills [13]. Still, most basic CS programming courses are still heavily lecture oriented with labs for the more hands-on activities. Assessment methods vary, but are typically based on exams, programming assignments, and projects.

Another active issue in CS programming education is concern about high barriers to entry and high failure rates in basic programming courses. Systematic studies of student "pass rates" - the percentage of students who complete a course with a passing grade, such as [14] and a recent follow-up study in [15] - found average pass-rates of about 67% in CS1 courses worldwide, with almost no change in the ten-year period covered by the studies. Ambrosia and Costa [16] conducted a study of PBL methods combined with other changes in a CS1 course in Brazil, a country with significantly lower-than-average pass rates. The observed pass rates improved from a several-year average of about 55% to about 80%. Despite some acknowledged challenges to validity and generalizability, studies such as this show promise for active-learner instruction models.

There have been attempts to determine if there is some type of 'intrinsic' programming ability in some students, which is not present in others. The most infamous of these, the unpublished but much-cited work of Dehnadi and Bornat [17], purported to separate programmers from non-programmers in terms of a simple test. In fact, Bornat [18] has since retracted any claim of efficacy, but there is still interest in the topic, positive or negative to the conclusion. A number of theories have been advanced to explain Dehnadi's observations, ranging from a willingness vs. unwillingness to accept that a machine will follow meaningless rules, an intrinsic ability to make consistent mental models, as well as prior work on the relationship of programming ability to understanding of the domain of application of the program [19]. It is this last point that influences our thinking on how to distinguish programming as pure computer science from programming as application development. We believe that knowledge of an application domain and its users is very important to learning to develop systems for that domain. This would seem to be obvious, but also seems to not be universally agreed upon.

B. Programming Outside Computer Science and Engineering

For at least 30 years educators and researchers have been concerned with how to teach programming to students outside of its 'native' disciplines of computer science (CS) and engineering [20]. To date, however, very little empirical research has explored the learning designs most effective in enabling non-traditional CS students to become effective professional software developers [21]. Groups outside of CS who attempt to learn programming include application developers, business and MIS programmers, web programmers, end-user programmers, and so-called 'conversational' programmers. Each of these different types present special requirements and challenges to those tasked with teaching them. A variety of factors have been identified as contributing to the success of non-computer science majors in introductory programming courses. Among the most pervasive and important of these is prior programming experience. Student math background, individual learning style, approach to organizing knowledge, and different conceptions of self-efficacy are also relevant [22]. Indeed, some research has shown that students from outside of engineering and science majors often lack a clear motivation for taking programming or other introductory computer science courses [23].

Perhaps the oldest category of non-CS programmer is that commonly referred to as application programmers. In the

traditional dichotomy of systems versus application programmer [24], CS graduates are considered most qualified as systems programmers, while programmers educated outside of CS were considered best qualified as application programmers. Systems programmers create the software that works directly with the computer, such as operating systems, device drivers, and other control systems designed to interface directly with hardware, while application developers focus on creating domain-specific software targeted to support some human activity.

A key skill in application development is the ability to translate domain problems into programs that correctly interpret and support needs given the goals, priorities, and operating environments of domain users. This kind of problem structuring and translation requires skills beyond those typically covered in a traditional CS programming curriculum. A focus on application domain challenges may be one of the keys to increasing the motivation of non-CS programming students. Students from outside of computer science have been shown to have a more positive experience in programming courses if the problems they encounter as part of the course are relevant to their primary field of study, in other words, their targeted application domain [23].

Similarly, web programming courses may help overcome motivation problems because many students are motivated to create exciting web experiences [25]. The melding of applications with infrastructure in web and mobile applications, however, adds significant complexity to the programming task. Some educators developing and evaluating web application courses have argued that requiring a typical CS1 course before web app development allows them to address significantly more interesting problems in the course [25]. Some work focused on courses in mobile application development skipped fundamental computing concepts altogether to move directly to creating GUI components and behaviors, and have shown some promise in increasing the motivation and confidence of non-CS majors [26].

Business programmers are a special sub-set of application programmers concerned primarily with creating transaction-based applications designed to help run organizations. The role of programming in the management information systems (MIS) programs intended to train these developers is widely debated. One study of MIS students [27] suggests they have very varied interests in programming and may only take one to two programming courses in their entire undergraduate career. In general, MIS graduates are not expected to be proficient programmers, though knowledge of the programmer's "thought process" and of at least one programming language are considered to be important as part of their skill set [28]. This same study found just over 12% of the program's MIS graduates were hired as programmers.

Most recently, the code academy model has emerged as a radically different alternative to the traditional university model. Students enter from diverse educational and professional backgrounds, the topics are oriented towards more vocational skills, the instruction is often interactive and hands-on, and programs are intensive to the point of complete immersion. A routine student comment is "drinking from a fire hose" or "being thrown in the deep end of the pool on the first day" [1]. According to a survey of most of the code academies in [1], Ruby, JavaScript, .NET, iOS, Python, and PHP account for about 98% of the dominant languages/platforms, with Java and Android development accounting for the rest.

The code academy emphasis is typically web development on one or two web development stacks. Assessment is based on completion of practical projects. Students are highly self-selected, learning goals are covered in a matter of weeks, and motivation appears to be very high. Opinions on the efficacy of code academies vary. They are new, out of the mainstream academic sector, and we could find no scholarly analysis of typical code academy pedagogy or efficacy. Much criticism centers on concerns about lack of training depth, exaggerated placement-rate claims, and the lack of a track record to assess them.

Another category of programmer trained outside of traditional CS programs are end-users themselves. Much of the research into end-user programming has focused on development of visual and other tools to support user creation of software, along with studies of using scripting languages such the Visual Basic scripting language embedded in Microsoft Office (especially Excel). Almost by definition, end-user programmers are those who learn to program by doing, in response to some perceived need for software capabilities they cannot otherwise obtain. We consider end-user programming beyond the scope of this paper.

Porter and Simon's [29] experiments with improving retention rates of students entering the CS major found three classroom innovations accounted for an 18% increase in students remaining in the CS major after the first programming course. These innovations focused on: contextualized computing (media computation), engaging classroom pedagogy (peer instruction), and supportive programming assignments (paired programming). One approach that has been shown effective is to have a CS 0.5 course when the students entering an introductory programming curriculum have diverse backgrounds. Students with no prior experience programming are to take this pre-programming course before entering the more intensive normal programming course offered to majors [30]. Some of these same innovations are likely to have similar positive effects on the success of other non-traditional students who seek to learn programming.

III. Learning Theory and Learning Programming

Learning pedagogy is frequently evaluated in terms of a revised version of Bloom's taxonomy [31] of educational objectives developed by Anderson and Krathwohl et al. [32]. This evaluates learning objectives, activities, and assessments in terms of a two-dimensional matrix of learning and cognitive dimensions (Fig. 1). The main goal of this section is to map important aspects of the training of different types of application developers to the learning and cognitive process dimensions in the taxonomy and draw conclusions about what type of training is essential for different types of development application domains.

Cognitive Process Dimension:

Learning Dimension:	1 Remember	2 Understand	3 Apply	4 Analyze	5 Evaluate	6 Create
A Factual knowledge						
B Conceptual knowledge						
C Procedural knowledge						
D Metacognitive knowledge						

Fig. 1. Revised Bloom's Taxonomy of Educational Objectives

An important question is how to map learning objectives and activities onto this matrix. Assessment is discussed separately in a later section. In general, programming and system development clearly bring all of the learning and cognitive process dimensions into play. From the very beginning, students must remember and understand (1-2) syntax and semantics (A-B) and apply (3) it to write (C) simple code fragments, algorithms, and programs. Although some analysis and evaluation (4-5) is required to do basic debugging and testing, most learning objectives, activities, and assessment focus on these basic (A-C) learning dimensions and cognitive process dimensions (1-3), regardless of application domain. Most beginning students have little knowledge of application domains, but some basic factual and conceptual application domain knowledge may be part of this early training. Thus different application domain targets may lead to some differences in course content and learning objectives for different basic programming courses.

As development training continues, students more deeply analyze and evaluate (4-5) to debug, test, and evaluate the efficacy of their programs for their domain. Increasingly this involves metacognitive knowledge of the domain (D) as well as the basic three learning dimensions (A-C). Ultimately, all learning and cognitive process dimensions become heavily in play, including creation using significant metacognitive knowledge of the domain of application, to create whole systems that do something useful in that intended domain.

One picture being painted here is that early programmer/developer training is fairly similar regardless of application domain. But another important question for a prospective 'application developer' is, "Which courses in a traditional computer science oriented curriculum focus on the higher level metacognitive learning dimension (D) and the higher-level analyze, evaluate and create cognitive dimensions (4-6) for the goal of system development?" Counting required CS core courses in the section on Programming in Computer Science and Engineering yields something like the following: background science and engineering (four courses); science and engineering math (four courses); basic programming (two courses); system internals (four courses), advanced theory, math, and algorithms (four courses), and advanced programming and software engineering (three courses), or about 20-21 courses total. It is generally agreed that the basic science and engineering (and its attendant math) plus the system internals courses are essential for a computer scientist's world-view, and comprise almost 60% of the CS core. But for an application developer, it's important to think about other critical issues that come up in developing systems designed for, e.g., business or personal use. These include business process flow, financial and accounting concerns, user requirements, usability of graphic user interfaces, security, and others considered in the next section. We conclude that some core advanced CS topics are not necessarily essential metacognitive and analysis, evaluation, creation knowledge for someone developing applications in business and personal domains. Of course, it is generally agreed that some of the standard advanced CS curriculum is important for an applications developer: discrete mathematics, algorithms and data structures, and software engineering, for example. But for other courses, there may be other training with significantly higher marginal utility to the training goals for an applications developer.

The point of this basic analysis of a typical CS program is that much of the required curriculum is not centered on building applications for non-science/engineering application domain users. This leaves the question of which traditional CS topics are essential for application developers, and what other topics outside these are most useful? The next section looks to recent ACM/IEEE Joint Curriculum Task Force's recommendations for insight.

A. ACM/IEEE Curriculum Task Force Recommendations

Up until 2001, the ACM/IEEE recommendations for computing curriculum [3, 33] were aimed at computer science programs. By 2005 [34] however, the computing curricular recommendations were broken out into five separate computing disciplines – using the ACM/IEEE notation and acronyms: computer engineering (CE), computer science (CS), information systems (IS), information technology (IT), and software engineering (SE). Each area was mapped onto a matrix/graph with five fundamental features on one axis (going from most hardware-oriented to most organizationally-oriented: computer architecture and hardware, systems infrastructure, software methods and technologies, application technologies, organizational issues and IS), and a continuum of development issues on the other axis going from more theoretical on one end (theory, principles) to more applied on the other end (application, deployment). Recognizing some overlap in areas, CE mapped largely onto hardware and systems infrastructure at all levels of theory and application, CS mapped largely onto system infrastructure, software methods, and applications mostly at the theoretical end, IS mapped onto organizational issues at all levels, and application technologies, software methods, and systems infrastructure at the application end, IT mapped onto organizational issues, application technologies, software methods, and systems

infrastructure mostly at the application end, and SE mapped onto systems infrastructure, software methods, and application technologies at all levels, with particular emphasis on software methods.

The specific curricular recommendations of this 2005 and subsequent area-specialized ACM/IEEE reports are complex, but can be summarized as requiring a foundation in these ten critical elements (somewhat simplified): discipline underpinnings; computer programming; limitations of technology; system lifecycle; concept of a process; area-specific advanced topics; interpersonal & management skill; exposure to application areas; professional, legal, & ethical issues; and integration of experiences via capstone project course.

Within this common foundation there are differences in emphasis for each of the five computing areas identified. CE, CS, and IS followed a continuous flow from previous recommendations because they historically have relatively large numbers of programs with a relatively long history. On the other hand, IT and were described as "the new kids on the block" in the US because they are relatively recent responses to a perception that new capabilities were needed but not handled by standard CS or IS programs. For example, describing IT, they wrote: "IT programs exist, not because CS or IS programs failed to do their job, but because those disciplines each define themselves as having a different job." Similarly describing SE: "The development of SE is a response to a very real problem: a shortage of degree programs that produce graduates who can properly understand and develop software systems. CS programs have shown that they can produce students who have sound skills in programming fundamentals. However, many believe that they have not been successful at reliably producing graduates able to work effectively on complex software systems that require engineering expertise beyond the level of programming fundamentals. "

A somewhat simplified view of the important differences between these five ACM/IEEE disciplines, ordered from more-technical to less-technical requirements, is given by:

CE: growing out of electrical engineering; strong physical science, engineering, math focus; little emphasis on software development, application technology, or organizational issues.

CS: less emphasis on physical science & engineering; strong math focus; strong emphasis on software development but focus on system internals; less focus on application technology or organizational issues.

SE: little emphasis on physical science & engineering; medium math focus; strong emphasis on software development but focused on applications, the software development lifecycle, methods, standards, testing, and quality control.

IT: little emphasis on physical science & engineering; little math focus; medium emphasis on software development but focus on practical technology issues, networking, database management, project management and planning.

IS: zero emphasis on physical science & engineering; little math focus; medium emphasis on software but focused on organization/user software like databases, user-centric systems, information management.

The remainder of the section focuses on the two new areas to which this paper is largely directed – SE and IT. One would get the impression from reading the ACM recommendations that programs in even the more technical of the two areas, SE, would significantly deemphasize typical physical science and engineering training and re-focus mathematics training from the continuous mathematics needed in physical sciences and engineering such as calculus, differential equations, and linear algebra towards more discrete mathematics. But the only two exemplars given in even the latest report from 2014 [35] show the standard three courses in science/engineering calculus, two terms of calculus-based physics, general chemistry with lab, an additional physical science course, and in one of them, matrix analysis and ordinary differential equations. This is the standard math/science background for most engineering programs. We concede the utility of mathematics and physical science in general. But there are limits to the number of required courses in a major and other areas like quantitative analysis for business, accounting and finance, economics and econometrics, operations research and optimization, planning/scheduling, supply-chain analysis, psychology of users and user interface design and areas may be more useful for many application developers.

The latest ACM IT recommendations from 2008 [36] show even reduced physical science and mathematics requirements than for SE – calculus was rarely mentioned and science focused mostly on "the scientific method". Although no curricular exemplars are given, even a cursory review of IT programs in the US indicates that although some programs still do require calculus, physics, or other lab sciences, these requirements are generally at a lower level.

We conclude from this review of ACM/IEEE recommendations that there can be a significant reduction in purely technical CS training for a software development program designed for domain applications outside of engineering and computer science.

IV. Teaching Programming for Information Sciences and Technology

Penn State's College of Information Sciences and Technology (IST) is part of the "i-school" movement, and graduated its first Bachelor's degree class in 2003. The intellectual range of the college is very broad, with faculty ranging from engineers and computer scientists to psychologists, sociologists, and business school graduates. There are several Bachelor of Science options in two majors, including people organizations and society, IT integration and application, information and cyber security, intelligence analysis and modeling, and design and development. The College has offered an Application Design and Development (D&D) option since its inception. Up until a few years ago, the option's emphasis was on high-level design, requirements planning, user interfaces, and project management. Programming fundamentals, data structures and algorithms, software engineering, and application development and implementation were not emphasized. Changes to the

curriculum were proposed by the authors about five years ago and approved in 2014. These changes increased attention to programming fundamentals, application of data structures and algorithms, object-oriented design and programming, software engineering, and agile and test-driven development.

As a result, the D&D option of the IST BS degree now maps closely to the ACM recommended curricula for Software Engineering (for the specific option requirements) and Information Technology (for the underlying major requirements). Option focus is entirely on development for application domains like business and web development and away from system internals. Math requirements, discussed later, are relatively lower than those for a typical CS program. The primary programming language is Java, not C/C++. The emphasis on data structures and algorithms is on how to assess their efficacy and complexity and how to properly select and use them in practice. PBL is used throughout the curriculum, but the first two programming courses focus on individual skill development, even if learning some skills in pairs or teams. But beyond those two courses, much work is done in pairs and teams.

The option emphasizes real-world application domain problems from the very beginning. This is done starting with the first required survey course on IST, the two beginning programming courses, as well as early courses on data management, databases and networking – all during freshman year. Examples are tailored to business and user-centered applications, and scientific/engineering applications are deemphasized.

The focus of the first applications programming course is to build a basic mental model of how computers and computer programs work, compile and run programs using basic data types, type safety, promotion and casting, variables, Java operators, Boolean decision-making and control-of-flow, loops, arrays, methods/functions, scoping rules, very basic exception handling and file input/output, and not much else in the way of programming concepts. This course carefully develops basic programming foundations and enforces good programming habits like naming variables, indenting blocks, simple program debugging and testing, and modular thinking. However, the application domains are decidedly non-science/engineering oriented. Most examples and exercises are business and user-centric.

The second applications programming course, specifically designed for D&D students, focuses on object-oriented programming, basic event-driven, graphic user interface development, finally basic data structures and algorithms. Again the focus is mainly on business and user-centric applications. Data structures and algorithms are explored mostly from the point of view of the Java Collections Framework and frequently makes use of built-in methods for common tasks such as sorting and searching. Instead of writing their own data structures like linked-lists and algorithms like binary search, students focus on using them in applications. Instead of the theory of complexity, students run simple experiments to see how different data structures and algorithms scale when the data set size increases. This and succeeding courses strongly emphasize defensive design and design-by-contract, and introduce test-driven development.

As students progress through the option, the focus widens significantly from strictly technical topics. Object-oriented design is treated as a separate course, and also covers interacting with users, requirements analysis and developing practical use cases. UML and CASE tools are used to study user interactions, class structure, as well as system architecture and deployment. Students study software lifecycle development models such as waterfall, iterative and incremental, spiral development, and a range of agile methodologies. In parallel, all IST majors take a course on "the user" – basic principles of the psychology of users, how users interact with systems, and how to study the needs of users.

The option also includes two-course D&D studio design sequence to give students a chance to practice different approaches learned in more fundamentals-based courses. Each D&D student must take at least one of these studio courses, and many take both. These studios are the subject of a concurrent paper at this conference [37].

At the senior level, there are three core courses in software engineering of large systems, user interface design, and distributed object computing, plus additional courses in web development and programming languages as they are available. Software engineering is typically a project-based course covering the entire software development lifecycle, but given what is done in earlier courses, especially focuses on agile development methodologies, formal unit-testing and more extensive test-driven development, quality assurance, and project estimating and management. Distributed object computing concerns itself with the basics of concurrent programming (especially transactional systems), basic network and web programming, client-server and peer-to-peer systems, as well as database transaction processing and an introduction to middleware. The user interface design course is more advanced study of users, user-centered design principles, and design of user interfaces. Each D&D student must take at least two of these three senior core courses.

This core development program give a total of two required core programming courses and four required advanced application development courses. Beyond this, there are six core IST required courses (IST survey, data management and databases, networking, information in organizations, analysis of users, senior capstone project), three math/statistics courses, one business or technical writing course, and one core economics course, for a total of 17 prescribed core courses. This leaves room for three (required) to five support-of-option courses, which can be taken from application domain areas in IST or other areas such as business, economics, psychology, mathematics, computer science, or engineering.

V. Lessons Learned and Way Forward

Because of the use of PBL, most IST D&D faculty interact closely with students, both in class and out. Thus some of our "data" are qualitative, based on close interactions with students over their four-year tenure. Based on this and results of other assessments – tests, sample programs, and larger projects – we

believe that senior-level students have significantly improved development skills, both in system specification and implementation since the first new courses were implemented about five years ago. Students are also increasingly succeeding in competitive development job interviews at top companies. Despite these improvements, there is still work to do. This section describes some of our experiences, issues confronted, and where we believe changes are needed.

A. Mathematics requirements

Many students arrive in IST without strong math backgrounds, at least relative to the typical math preparation of a physical science or engineering/CS major. Most are not motivated to study math developed for physical science and engineering problems, which should not be surprising since they are not planning to study physical science or traditional engineering fields. Currently, three core courses are required for all IST majors, regardless of option: calculus, basic (mostly descriptive) statistics, and discrete mathematics. The first two do not focus on applications useful to many of these students. The third, discrete math, has a high topic density and prior math training does little to prepare them for the level of abstraction required. The high topic density also leaves little time to explore appropriate applications of discrete math.

One of the main issues is the one-size-fits-all approach to math training. This is common in many technical disciplines, and is apparent in ACM/IEEE sample programs, e.g., [34-35] and even a cursory sample search of typical requirements of university computing programs. But this can become a serious issue in a college as intellectually diverse as IST. It should be clear that D&D students need more math training than someone studying, e.g., people, organizations, and society. D&D students in particular need to slowly and carefully develop the critical foundations in critical thinking, deductive and inductive logic, sets, relations and functions, sequences, recurrences, and recursion, combinatorial analysis, discrete probability and statistics, number theory, graph theory, algorithms and data structures, basic algorithm analysis, language theory, and computational modeling, and then apply these ideas in their application programming and development courses.

So it seems clear that the current mathematics focus in many application design and development curricula, including ours, would benefit from a significant shift away from physical science and engineering math courses and towards much more discrete math, algorithms, and data structures. This is consistent with the ACM/IEEE recommendations in, e.g., [34-35]. Our students also tell us that lack of sufficient breadth and intensity in discrete math, algorithms, and data structures can be a major issue at competitive interviews at top development companies, not to mention when approaching further study in graduate school. In fact, several top students have themselves proposed a new applied algorithms and data structures course to fill this need. D&D faculty enthusiastically support this.

B. Agile and Test-Driven Development

As pointed out in the 2005 ACM/IEEE computing curricula recommendations [34], application developers and software engineers need significantly more focus on the application development process than is done in a typical computer science curriculum. Two important ideas have emerged from current application development practice. First, students need to learn the discipline of design-by-contract and adherence to lightweight, agile development methods with closely-spaced but firm deadlines and rigorous code reviews to show progress on projects and avoid drift. Second, students need to develop the discipline of test-driven development and unit testing small units of code as they go. In fact, these two approaches strongly support each other. This is primarily done in later courses, but the intent is to begin pushing these ideas earlier in the curriculum.

C. Problem-Based Learning (PBL)

The College of IST started out emphasizing PBL in its teaching methodology from the beginning. Learning to program requires constant practice and a great deal of time-on-task. PBL supports in-class practice under the watchful eye of instructor and assistants. Our experience is that properly managed individual and pair in-class programming sessions can significantly reduce student fear and confusion in early programming courses. In more advanced courses, where students now have the basic core programming skills in-hand, PBL is used more to develop team software development skills. Using PBL, lecturing is used to a minimal extent to scaffold critical ideas and answer questions that come up. Then the maximum amount of time possible is available to coach students as they work on programming/design or math exercises and projects. Potential barriers to this approach are a perceived lack of relevance, a lack of intensity, and insufficient preparation or scaffolding of critical knowledge by students.

Interestingly, code academies typically embrace these principles – immersive learning, an active-learner environment, relevant and practical problems that engage students, and close interaction between students and faculty. This is consistent with our experience that students learn skills like programming, design, and mathematics better in such an active-learner environment. Future goals are to continue to make the D&D curriculum more relevant, eliminate unreasonable barriers, but double-down on critical foundational preparation so students can better handle abstraction and problem-solving approaches.

D. Assessment, Learner Resilience, and Self-Efficacy

Assessment of non-CS major beginning programming students can be a challenge. There are a few online programming practice systems in use, primarily to learn basic concepts and syntax. Most if not all instructors give weekly sets of problems and programming exercises. Some are done in class with coaching, others out of class. Many instructors have some type of class discussion forum where students can ask questions any time of day or night - students are encouraged to use it, and instructors or TAs respond quite rapidly to questions. But performance on out-of-class exercises has not necessarily been a good predictor of individual programming skill. Thus in-class programming tests that, in total, make up a significant percentage of each student's grade are an important part of assessment. Sometimes this is

questioned in a PBL context. However PBL is an active learning approach, not an assessment method. The well-known earliest applications of PBL were in training medical doctors. To our knowledge, nobody has ever suggested that doctors shouldn't be required to pass rigorous licensing and board examinations simply because they learned medicine using PBL.

Writing computer programs on a timed test can be challenging for a beginning programming student. Many modern college students are used to multiple-choice tests of declarative knowledge and are not accustomed to performing technical tasks like programming on an in-class timed test. Many students tell us they dread these tests, even when given frequently for low stakes or with repetition permitted without penalty. So this type of rigorous assessment approach does frustrate some students and also makes a high grading load. Automated compilation and execution of electronically-submitted student programs against test cases can help with grading load. But there is no substitute for carefully reviewing each student's code and giving rapid and fairly detailed feedback. This is mostly an issue in larger sections – in small sections, it is often possible to accurately and fairly rapidly assess students on-the-spot during class. This hands-on approach is far preferable to giving formal written tests, but has not scaled beyond 20-30 students per section, even with assistant support.

Beyond these practical issues, some are concerned that rigorous assessment can lead to discouragement and low self-efficacy in the sense of Bandura [38]. In [22], Wiedenbeck studied self-efficacy in the context of a first programming course for non-majors. The results were preliminary, but the conclusion in [22] was that high prior self-efficacy was not necessarily a good predictor of performance, but that knowledge organization was. This fits with Bandura's model, which argues that students with low prior experience may actually overestimate their capabilities, leading to poor results. Instead, [22] argues that slow, steady knowledge building and organization led to better performance results. This fits with our experience that slow, deliberate building of concepts, careful assessment, and rapid feedback are essential.

E. Changing Landscape

The emergence of big data and the complexity of data analytics is increasing pressure on non-traditional programming students to understand the intricacies of data structures and algorithms. Whereas in the past educators could reasonably focus on approaches to evaluation and selection of pre-existing structures, it may be more important than ever for all programming students to understand the design, rationale, and implementation of sophisticated data manipulation methods. There is also an ethical dimension to understanding algorithms since they are so pervasive and potentially consequential in our day-to-day activities.

Similarly, the evolving "internet of things" is increasing interest in a sound understanding of heterogeneous application programming interfaces and low-level hardware device drivers. These are areas that have been outside the scope of what an application developer needs to do their work. For example, some of our development students are very interested in projects using Arduino and Raspberry Pi platforms to do projects in many application areas. This doesn't necessarily imply that all applications developers should suddenly be required to take a full complement of CE/CS courses on system internals. Our D&D students who are motivated in this direction seem to have no problem pursuing the required knowledge, provided they have clearly mastered basic programming and development principles.

VI. CONCLUSION

Both prior literature and our experience indicate that many factors influence the ability of non-traditional students to learn programming and application development skills. From the student point of view, these include their degree of passion for learning; their confidence and perception of self-efficacy when faced with a difficult problem [22]; the time and effort they are willing to invest learning; and how they handle abstract ideas and organize their prior programming knowledge, including common solution plans and patterns. From the instructor point of view, these include the relevance of the problems and exercises students work on as part of a course [23], including the ability to create exciting problems that capture their imagination [25-26]; the degree to which active experiences are used to enhance learning [16, 39]; and the ability to give appropriate and rapid feedback of performance without crushing students' confidence and enthusiasm for the subject.

Our experiences suggest that both PBL and active, collaborative learning are important pedagogical techniques for teaching application development, in line with the positive experiences reported in [16, 39]. But overuse of collaboration, especially in assessed work, without considering its context can be a double-edged sword by allowing students to potentially skirt difficult but important concepts and tasks. The objective should be to gently but firmly push students in the "correct" direction.

Finally, empowering learning in a design and development curriculum is informed by gathering data and applying relevant science. But success also involves important components of art and passion – from both students and faculty. These qualities are typically hard to measure, but generally recognizable when observed. And thus we would argue that a certain degree of flexibility and willingness to dynamically adapt teaching methodology is essential to capture that passion when it reveals itself.

ACKNOWLEDGMENT

Both authors thank Larry Spence and Fred Fonseca for useful discussions over many years about teaching and problem-based learning.

REFERENCES

[1] L. Eggleston. (2015, June 28). *2015 Bootcamp Market Size Study* [Online]. Available: https://www.coursereport.com/2015-coding-bootcamp-market-research.pdf

[2] L. Eggleston, L. (2015, October 26). *Course Report Bootcamp Graduate Demographics & Outcomes Study* [Online]. Available:

https://www.coursereport.com/resources/course-report-bootcamp-graduate-demographics-outcomes-study

[3] ACM/IEEE Joint Task Force on Computing Curricula (2001, December 15). *Computing Curricula 2001 – Computer Science* [Online]. Available: http://www.acm.org/education/education/education/curric_vols/cc2001.pdf

[4] ACM/IEEE Joint Task Force on Computing Curricula (2008, November 15). *Computer Science Curriculum 2008: An Interim Revision of CS 2001*. [Online]. Available: http://www.acm.org//education/curricula/ComputerScience2008.pdf

[5] ACM/IEEE Joint Task Force on Computing Curricula (2013, December 20). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Available: http://www.acm.org/education/CS2013-final-report.pdf

[6] A. Soares. "Problem based learning in introduction to programming courses". J. Comput Sci College, vol. 27, p 36, October 2011.

[7] J. O'Kelly and J. Gibson. "RoboCode & Problem-Based Learning: A non-prescriptive approach to teaching programming". In Proc. ACM ITICSE, vol. 11, pp. 217-221, September 2006

[8] C. Loftus, L. Thomas, and C. Zander. "Can graduating students design: Revisited". In Proc. ACM SIGCSE, vol. 42, pp. 105-110, March, 2011.

[9] R. Lingard and S. Barkataki. "Teaching teamwork in engineering and computer science". In Proc. IEEE FIE, pp. F1C-1 – F1C-5, October, 2011.

[10] [A. Begel and B. Simon. "Novice software developers, all over again". In Proc. ACM ICER, vol. 4, pp. 3-14, September, 2008.

[11] [A. Begel and B. Simon. "Struggles of new college graduates in their first software development job". In Proc. ACM SIGCSE, vol. 40, pp. 226-230, March, 2008.

[12] A. Radermacher and G. Walia, "Gaps BetweenIndustry Expectations and the Abilities of Graduates: Systematic Literature Review Findings". In Proc. ACM SIGCSE, vol 44, pp. 525-530, March, 2013.

[13] R. Vivian, K. Falkner, N. Falkner, and H. Tarmazdi. "A Method to Analyze Computer Science Students' Teamwork in Online Collaborative Learning Environments". ACM Trans. Comput. Ed., vol. 16, article 7, March, 2016.

[14] J. Bennedsen and M. Caspersen. "Failure rates in introductory programming". SIGCSE Bulletin, vol. 39, pp. 32-36, June, 2007.

[15] C. Watson and F. Li. "Failure Rates in Introductory Programming Revisited". In Proc. ACM ITiCSE, vol. 19, pp. 39-44, June, 2014.

[16] A. Ambrosio and F. Costa, "Evaluating the Impact of PBL and Tablet PCs in an Algorithms and Computer Programming Course", In Proc. ACM SIGCSE, pp 495-499, March, 2010.

[17] S. Dehnadi, S. and R. Bornat. (2006, February 22). "The camel has two humps (working title)". Available: http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

[18] R. Bornat. (2014, July 24). "Camels and humps: a retraction". Available: http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf

[19] B. Adelson and E. Soloway. "The role of domain experience in software design". IEEE Trans. Softw. Eng, vol. 11, pp. 1351–1360, November, 1985

[20] K.M. Galotti and W.F. Ganong. "What non-programmers know about programming: natural language procedure specification". Int. J. Man-Mach. Stud., vol. 22, pp. 1-10, January, 1985.

[21] P.K. Chilana, C. Alcock, S. Dembla, A. Ho, A. Hurst, B. Armstrong, and P.J. Guo. "Perceptions of non-CS majors in intro programming: The rise of the conversational programmer." In IEEE Symp. Vis. Lang. Human-Centric Comput., pp. 251-259, October, 2015.

[22] S. Wiedenbeck, S. "Factors affecting the success of non-majors in learning to program". In Proc. ACM Int. Wkshp. Comput. Ed. Res., vol. 1, pp. 13-24, October, 2005.

[23] S. Kurkovsky. "Making computing attractive for non-majors: a course design". J. Comput Sci College, vol. 22, pp. 90-97, January, 2007.

[24] J. Ousterhout. (1998). "Scripting: Higher Level Programming for the 21st Century". IEEE Computer Magazine, pp. 23-30, March, 1998.

[25] M. Stepp, J. Miller, and V. Kirst. "A CS 1.5 introduction to web programming". In Proc. ACM SIGCSE, vol. 40, pp. 121-125, March, 2009.

[26] W.L. Honig. "Teaching and assessing programming fundamentals for non majors with visual programming". In Proc. ACM ITiCSE, vol. 18, pp. 40-45, July, 2013.

[27] A.D. Ritzhaupt, T.G. Gill. "A hybrid and novel approach to teaching computer programming in MIS curriculum". In Negash, S., Whitman, M., Woszczynski, A., Hoganson, K., & Mattord, H. (Ed.), *Handbook of Distance Learning for Real-Time and Asynchronous Information Technology Education*, Hershey, PA: IGI Global, 2008.

[28] I.C. Ehie. "Developing a management information systems (MIS) curriculum: perspectives from MIS practitioners". J. Ed. Bus., vol 77, pp. 151-158, January-February, 2002.

[29] L. Porter and B. Simon. "Retaining nearly one-third more majors with a trio of instructional best practices in CS1". In Proc. ACM SIGCSE, vol. 44, pp. 165-170, March, 2013.

[30] R.H. Sloan and P. Troy. "CS 0.5: a better approach to introductory computer science for majors". In Proc. ACM SIGCSE, vol. 39, pp. 271-275, March, 2008.

[31] B. S. Bloom (Ed.), M.D. Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl. *Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive domain*. New York, NY: David McKay, 1956.

[32] L.W. Anderson (Ed.), D.R. Krathwohl (Ed.), P.W. Airasian, K.A. Cruikshank, R.E. Mayer, P.R. Pintric, J. Raths, and M.C. Wittrock. (2001). *A taxonomy for learning, teaching, and assessing: A revision of Bloom's Taxonomy of Educational Objectives (Complete edition)*. New York, NY: Longman, 1956.

[33] A. B. Tucker, ed. (ACM/IEEE Joint Task Force on Computing Curicula. *Computing Curricula '91*. IEEE Computer Society, June 1991.

[34] ACM/IEEE Joint Task Force on Computing Curricula (2005, September 30). Computing Curricula 2005: The Overview Report covering undergraduate degree programs in Computer Engineering, Computer Science, Information Systems, Information Technology, Software Engineering. [Online]. Available: http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf

[35] ACM/IEEE Joint Task Force on Computing Curricula (2015, February 23). *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Available: http://www.acm.org/binaries/content/assets/education/se2014.pdf

[36] ACM/IEEE Joint Task Force on Computing Curricula (2008, November). *Information Technology 2008: Curriculum Guidelines for Undergraduate Degree Programs in Information Technology*. Available: http://www.acm.org//education/curricula/IT2008%20Curriculum.pdf

[37] S.R. Haynes and D.R. Mudgett. "A Design Studio Course in Application Development: Lesson Learned". IEEE FIE, October, 2016, submitted for publication.

[38] A. Bandura. "Self-efficacy". In *Encyclopedia of Human Behavior*, Vol. 4, V.S. Ramachaudran, Ed. New York, NY: Academic Press, NY, 1994, pp. 71-81.

[39] Poindexter, S. (2003). "Assessing Active Alternatives for Teaching Programming", Journal of Information Technology Education, Volume 2.