Designing the Ideal Assessment System to Support Mastery Learning of Computer Programming in an Online Environment

Steven Shaffer, Martin Yeh, Thomas Iwinski
Penn State University Park

In this paper, we describe the design of our fourth-generation interactive system for teaching computer programming courses based on the principle of mastery learning. Previous versions of this system have been used since 2005, with published papers describing the results. Our updated system includes new features (i.e., assessing students' programming abilities) that are based on a decade of research and experience delivering online programming courses.

Traditional distance education courses enabled location and time shifting, but created a burden of delay in providing students timely feedback, interaction, assessment, and evaluation of performance. With proper software support, we can still accommodate location and time shifting, and now have the capability to eliminate the delays in feedback, interaction, assessment and evaluation. The shortening of these feedback loops in the instructional process greatly aids student learning.

Students need lots of practice to learn to program such that the volatile declarative knowledge can become more robust procedural knowledge. However, many students will not do assignments unless there are some "points" associated with doing them. If there are points involved, then someone will have to grade the programs. Some online programming classes can typically have hundreds of students; if 30 assignments are given throughout the semester, this amounts to 3000, 6000 or more programs to grade. That's a lot of programs to grade, no matter how many TAs there are. Thus, for students of large online courses to get the practice that they need, some sort of grading automation is necessary.

A common problem in computer science courses is the "cascading misunderstanding" phenomenon. If a student misunderstands a concept in an early course, s/he may still pass and continue into later courses, compounding the misunderstanding semester after semester. A solution to this problem is a switch to a mastery learning model. With mastery learning, a student cannot move to module n+1 without first having mastered module n. This is achieved by continuing to allow the student to attempt assessments in module n until mastery is achieved. These continuing attempts may also require additional intervention to support student learning. This means that any particular assessment might be formative or summative, depending on whether or not the student completed it properly. Although not a panacea, this approach can go a long way toward enhancing student success.

We implement a mastery learning model within our course, supported by our in-house developed assessment software. Our assessments are both formative and summative, and thus mastery learning is intricately woven within the fabric of the entire course design.

Assessment security, authorization, and plagiarism detection are particularly difficult problems in an online learning environment, where proctored assessments are difficult to manage. If students cheat and get away with it, this diminishes the viability of the course, the program, and the degree which is conferred. Our system also includes mechanisms to detect and eliminate these types of issues.

I. Introduction

Using software to facilitate learning to program has a long and rich tradition in computer science education; [8] describes this history in detail. Student research studies by the first author of this paper can be found in [8] and [12] ; the reader is referred to these papers for the appropriate background and results of that human subjects research. This paper adds to this prior work by reporting on the next phase of designing the ideal mastery learning / assessment system for computer science courses.

The field of online learning has grown exponentially during the past two decades, and is now firmly entrenched in higher education. For example, [9] summarizes the organizational, technical and legal issues of assessment systems used for online learning.

Combining the results of research in these traditions, along with commercial applications and the pragmatic, "in the trenches" development experience of the authors, enables the possibility of positing the ideal design for an online learning assessment environment for programming classes.

Traditional distance education courses facilitated learning by permitting location and time shifting of the instructional process and afforded accessibility to students who may not have access to a residential learning experience. This also created a burden of delay in feedback, interaction, assessment, and evaluation of performance. With distance education moving to an online environment we can still accommodate the location and time shifting, and now have the capability to eliminate the delays in feedback, interaction, assessment and evaluation. The shortening of these feedback loops in the instructional process greatly aids student learning [1].

II. Context

Teaching programming via online learning is a multi-dimensional problem. Course content is of course an important issue, but this is outside of the scope of this article, except to say that it is generally unacceptable to change course content simply because the delivery will be online. The choice of programming language

(e.g., C++, Java, Alice, Scratch, or Processing) is also important, and is often the fodder for sectarian wars among faculty. Let us assume, though, that the choice of language should also not be affected by the delivery platform.

Online learning programming course designs run the gamut from "read the book and answer a series of multiple choice quiz questions" (a horrible way to learn to program) to sophisticated designs using video, interactive textbooks, and uniquely designed, powerful, integrated development environments (IDEs). Selecting the right combination of the large array of potential technologies to optimize the delivery of a programming course online is a complex problem, where simplicity for teaching novice students is greatly needed.

Programming is a skill more than it is a knowledge set [10], and thus learning programming involves practicing programming. How will your students learn to program? The problem with static text examples (whether a hard copy or an online book) is that they do not demonstrate the process behind the task of programming. Sample solved problems give students the impression that programs are written from the top down (how they read them), or are somehow handed down from God on stone tablets. Fixed hardcopy textbooks are limited to presenting text and static images; whereas well designed online course allows for linear and interactive animations as well as virtual instruments and tools that permit data input/output that demonstrate the process of programming in a way that is impossible for any static text.

How will your students practice programming? Will they simply re-type the examples from the textbook (or worse, copy/paste them from an online text)? Using a professional IDE has the advantage of ecological validity, but may result in an excessive cognitive load [13] for the student. Does an introductory student really need to select one of the seven different project types, just to create a "Hello, World!" program? IDEs that minimize cognitive load without needlessly restricting student development decisions hold the best of both worlds.

III. Focus on assessment

As important as these factors are, this paper focuses primarily on assessment, specifically regarding how proper use of assessment technology can support student learning. Our assessments are both formative and summative, as described below, and are thus intricately woven within the fabric of the entire course design.

How will you assess your students and provide meaningful feedback in a timely manner? Multiple-choice is a notoriously bad way to test someone's programming skill. Given that an assessment requires actual programming, how much programming will be required? Solutions to this have run from fill-in-the-blank assessment tools (where the student is presented with an entire program and is tasked with filling in one or several blanks to make it operate) to large scale programs which the student will write and debug in a professional IDE and submit to the instructor for testing and grading. Of course, the answer to this

question depends on the nature of the instructional content, the student's ability level, and the delivery environment. Thus, different courses will need different solutions.

IV. Assessment security

Assessment security, including student identification, authorization, and plagiarism detection are particularly difficult problems in an online learning environment, where proctored assessments are difficult to manage. There are now available web sites where a student can hire someone to take an entire class in the student's place; how do you know who is doing the programming, and, even if it is the students themselves, how do you know if they are sharing answers?

Assessment security is especially important in the wake of The Higher Education Opportunity Act, which states in part that the accreditation agency or association must require "an institution that offers distance education or correspondence education to have processes through which the institution establishes that the student who registers in a distance education or correspondence education course or program is the same student who participates in and completes the program and receives the academic credit"[5].

Identifying who indeed is taking an assessment can be complex. At a minimum, a webcam can be used to attach a photo of the student to the submission, to be compared to an official ID photo. This can be easily circumvented though, by having the student sit for the photo, then let her friend take over doing the assignment. Including a constant video feed with the assignment would probably take up too much space and bandwidth. Contract proctoring services will watch the student throughout the process for a fee, but this can get expensive.

Authorization – checking that users do not have access to parts of systems or data to which they should not have access – can be complex in some contexts, but for assessments systems usually the problem is limited to: (a) identifying who is an instructor and who is not, and (b) restricting access to course material for which the user is not associated. Careful design of access control lists will go a long way toward solving these issues.

Security can be a major issue with an online program submission system, precisely because students are executing program code, possibly on a server. Potential problems can occur from accidental or purposeful misadventure. For example, simple infinite loops are common in student code, and without a method of interrupting these, a server could quickly get bogged down. As an example of purposeful mischief, in one early version of our work, a student submitted the following line of code (a "fork bomb"):

```
while (true) fork();
```

Which quickly crashed the server. There are several approaches to blocking this type of attack, which are discussed later in this paper.

V. Other forms of cheating

If students cheat and get away with it, this diminishes the viability of the course, the program and the degree which is conferred. As many as 75% of college students admit to at least some cheating, with between 19% and 38% of students admitting to frequent cheating [6]. In addition, cheating is easier in a technology-rich environment [4], and thus simply ignoring this possibility is not an option. In addition to the methods described above, students can participate in a number of other cheating approaches in an online programming course:

Online "help" sites: Several online sites exist where students can submit their homework problems and get solutions. These "helpful" folks are usually experienced programmers who don't realize that if students don't learn to think on their own, there won't be any experienced programmers in the future.

Posting / finding posted solutions on the web: Other "helpful" students will post their solutions to problems on the web for future course-takers to use. This practice is an extension of the common fraternity practice of storing old exams from courses for later students to use. The difference now is that one post can reach thousands of students.

Accessing references not allowed during assessments: Although programming instructors often don't care what language reference student's use, this could be an issue if the problem comes straight from a well-known textbook.

VI. Pedagogical model

Students need lots and lots of practice to learn to program such that their volatile declarative knowledge can become more robust procedural knowledge [7]. However, many students will not do assignments unless there are some "points" associated with doing them. If there are points involved, then someone will have to grade the programs. Some online programming classes can typically have hundreds of students; if 30 assignments are given throughout the semester, this amounts to 3000, 6000 or more programs to grade. That's a lot of programs to grade, no matter how many TAs there are. Thus, in order for students of large online courses to get the practice that they need, some sort of grading automation is necessary.

Another pervasive problem within computer science courses in general, and online learning courses in particular for the purposes of this paper, is what we refer to as the "train wreck" problem. This occurs over several semesters, where students are taking a series of courses in a prerequisite chain. For example, perhaps the student takes course CS1 and never really quite grasps the concept of variable scope. However, it is a small part of the grade, and the student gets an A-in the course. Now, that student moves to the CS2 course, where the lack of

understanding of variable scope affects his understanding of classes. But, this is still a small-ish part of the grade, and he ends up with a B in the course. In CS3, his lack of understanding of scope and classes interferes with his understanding of recursively-defined data structures, and now he only receives a low C in the course. As he moves to the last course in the sequence, he is hopelessly lost, and the instructor wonders how he ever got to the fourth course in the first place.

This "train wreck" of cascading errors could have been averted through a proper remediation of his understanding of variable scope in CS1. A solution to this problem is a switch to a mastery learning model [3]. With mastery learning, a student cannot move to module $n+1$ without first having mastered module n. This is achieved by continuing to allow the student to attempt assessments in module n until mastery is achieved. This means that any particular assessment might be formative or summative, depending on whether or not the student completed it.

Although not a panacea, this approach can go a long way toward enhancing student success [8]. The only thing standing between the student and a strong understanding of the material is perseverance (time on task) and the quality of the instructional materials.

VII. Designing the ideal assessment system

The primary components of an instructional process are content presentation, interaction, assessment and evaluation with frequent feedback loops to accommodate adjustments to the learning process. Some of these components are already supported by existing software tools. However, assessments are not well supported, especially with regard to programming. And, since the mastery learning model places high focus on assessments, it was important for us to fill this void.

We have had three earlier attempts at implementing software to support programming assessments for online learning, dating back to 2004 [12]. During these three earlier iterations, there have been many lessons learned. Now, based on those lessons, we have designed what we hope is the ideal solution to this problem. We say this somewhat tongue in cheek, as perfection is at best asymptotic, and at worst a matter of opinion.

The most significant component of such a system is a method to engage students in an interactive assessment. In general, an ideal programming assessment system will consist of a desktop controlled environment, which allows/enables just the right amount of user power without overwhelming the student with too many options. Multiple choices of programming language may be allowed, based on the course the student is enrolled in. The student should be able to request assignments, solve assignments, and submit assignments all from within the same interface. Ideally, solutions should be run through a gauntlet of tests before allowing submission. For example, the "gauntlet" might check for appropriate commented headers, and that the code compiled cleanly (perhaps even without

warnings). It might also run an input/output check for several data sets, checking for proper results. If the assignment contains a requirement such as "write a function named 'amortize'", then the pre-processing software could check for that. If the student's submission fails any of these tests, the student will not be able to submit the assignment, but will instead receive an instructional prescription regarding how to resolve the deficiencies found in the student's submission. The student keeps on trying until she succeeds.

Additional aspects of this idealized system might include the following:

Controlling access to the desktop: Once an assessment has begun, making sure that the student did not leave the assessment window is a good strategy. For example, this will keep the student from trying to do an internet search for a solution, or to use a reference that is off limits. This approach has some limitations in online learning, however, because students can use other computers to access material. This problem is ameliorated somewhat through the use of video proctoring, discussed below.

Time limits: Given infinite time, a student may stumble upon a solution to a problem even without an understanding of the material. Worse, infinite time might give the student the opportunity to consult a friend via email or post the assignment on one of several "helper" web sites to obtain an answer. Arguments can be made that "real world" programmers don't have arbitrary timelines; these arguments can be countered by (a) noting that everyone has deadlines, and that (b) true fluency requires timeliness. In our classes, we always strongly urge students to complete a problem, even if time has run out; this avoids the problem of student "thrashing" – spending more time switching between problems than solving them. In addition, the software will not serve a new problem to a student until the time limit for the last one has run out, thus avoiding student "fishing" for an "easier" problem. In any case, an ideal assessment system should allow for time limits, placed under the instructor's control.

No copy/paste: Students, especially novice students, are notorious for injudiciously using copy/paste in their programs. Thus, one might see dozens of lines from a tic tac toe program show up in a mortgage amortization program. Eliminating the ability to copy and paste reduces the occurrence of this pathology to nearly zero, since doing so involves extra work (typing). In addition, in the case that the student does receive unauthorized help with a program, he will at least have to type the program in himself, learning something about programming in the process. This could also be a configurable feature of the ideal system.

Forced indentation and formatting: Many programming languages ignore whitespace, and thus most IDEs allow students to create absolutely horrendous code. A useful aspect of an ideal assessment system would include automated/forced formatting. Although many professional programmers hate these, novices do not process programs like experts do [10], and thus constraining students to create readable programs has pedagogical value. Options could

include (a) post-processing rather than real-time formatting, and (b) instructor level control to turn off this feature.

Code analysis: In order to avoid certain types of malicious submissions, it is plausible to pre-process the code, disallowing access to certain libraries and features of the target language which could cause trouble. Although effective, this approach requires the development team to stay one step ahead of the nefarious designs of the ne'er-do-well student. As they say about terrorists, they only have to succeed once, while the good guys have to succeed all the time.

Another use for a code analysis step is to catch violations of internal requirements (header comments, function names, global variables, etc.). In lieu of specialized checkers for each submission, this requirement can be handled by allowing the instructor to submit regular expression checks for each assignment. Not all internal requirements can be checked with regular expression matching, but many can.

Logging every program execution: If every student program execution is properly logged before being executed, the author of any program which causes harm to the server environment can be tracked down.

Run in a sandbox: Running submitted programs in a sandboxed environment can be an effective counter-measure to malware. This approach is often used for coding contests, which is a similar situation to programming assessments.

Execute on the client: Another solution to the malware problem is to execute all code at the client level, capturing output and sending only the output to the aggregation server. This is probably the safest approach from a security standpoint, but it is very difficult to implement. For example, generation three of our systems actually implements a C++ interpreter inside of a Java program. As an interpreter, all attempts at malicious code can be interrupted. In addition, infinite loops are detected via a simple iteration counter: if the software came back to a particular line more than n times, the program probably has an infinite loop and it is aborted. However, implementing an interpreter is no small task, and that project was only successful because the target population (CS1 students) only needed a subset of the full C++ language.

Generate programming problems on the fly. One way to eliminate the problem of students having access to past programming problems is to never re-use them. However, as anyone who has created programming problems will tell you, it is no small feat to continue to think up new problems all the time. In addition, most online learning classes are asynchronous, and so someone would have to think up hundreds of equivalent problems ahead of time. Also, students need many problems to practice on, and as was said previously, experience tells us that they often will not do the problems if there are no "points" associated with turning them in. Our solution to this is to create an effectively infinite number of

problems on the fly, using banks of templated problem/solution descriptions. For example, the template:

> The formula for [CONCEPT] is [FORMULA]. Prompt the user for each of the variables, calculate the [NAME], then display the answer to the user.

Can be turned into the following problem specification:

> The formula for the area of a triangle is (base * height)/2.0. Prompt the user for each of the variables, calculate the area, then display the answer to the user.

And can be tested with the following, automatically generated, program segment:

```
int base;
cout << "Enter base: ";
cin >> base;
int height;
cout << "Enter height: ";
cin >> height;
double area = (base*height) / 2.0;
cout << "The area is " << area;
cout << "." << endl;
```

Admittedly, this is easier to do for simple problems that for complex ones, but it is the lower-level programming courses (CS1 and CS2) that have the largest number of students. We have implemented a robust system for generating thousands of equivalent programming problems.

In our course implementation, practice, assignments and assessments are one and the same. Students are served assignments of a certain level of difficulty repeatedly until one is solved. Once a problem of type n is solved, the student can move on to problems of type n+1. There is no penalty for the number of tries a student takes to solve a particular problem type.

Run the student programs before allowing the student to submit. As the student works on the program, he can test it against a published input/output data set. If the initial tests are passed, the program is run against a hidden set of input/output data. The results of passing these tests are returned to the user. If all tests are not passed, the student is not allowed to submit the program and the student is given an instructional prescription related to errors identified within their code or the general reason for the failure. This feedback system functions as a basic performance support system.

Capture photos at random intervals. As discussed, ideally a human would proctor a student taking an exam; however this is infeasible for a large online class. An alternative is to capture webcam photos and screen shots at random intervals, to

be packaged along with the student's submission. These can be quickly reviewed for violations, have relatively small bandwidth requirements, and serve to flag a majority of situations. If nothing else, awareness of the system will have a chilling effect on the potential rouge student.

Controlling student program execution on a server. If student programs will be run on a server, some best practices include: (a) running the programs in a queue to control system load, (b) setting external interrupts after (say) 1 second of execution time, and (c) always logging the student id and program before executing it, so that any malware can be traced back to the student. Note that student wait times will increase markedly for assignment responses in queued requests if every student has the same deadline. This is yet another advantage of using the mastery learning model, since students will be automatically be working at different paces.

The following items are under consideration for our ideal assessment system; they have been suggested but have not made the "cut" for the next version.

Capture audio as well as visual info. This was suggested in order to ascertain if the sounds of keys clicking matched what the student seemed to be doing.

Keystroke analysis. Typing rhythm can be used for identity verification [2], and may also be useful to highlight workflow interruptions, etc.

Post-hoc error analysis. There may be pedagogical value in developing reports and analyses of errors on a per-student basis. This might then be used to advise students with regard to remedial work.

Explicitly differentiating syntax errors from semantic errors. There may be a pedagogical advantage to requiring a two-step process for a student to run his code. For example, a "check" button could check for syntax and formatting errors; only if the check process worked, would the "run" button be available. The "run" button would then execute the student program against the input/output pairs and display the results to the student. The notion behind this two-step process is to reinforce to the student the difference between the syntax and semantics of a program.

VIII. Student user flow

Figure 1 shows the basic flow of a single student assessment using an idealized assessment system. After logging in, the student selects an assessment. The system responds by requesting a student ID be placed in front of the web cam and a photo is taken. Next, the student is asked to sit face-front to the web cam and another picture is taken. These are both logged to the server, at which time a unique programming problem is generated. A timer is started, and the student begins to work on the problem, obtaining feedback every time the program is run. Periodically, web images and screen shots are captured and stored locally. When the student has successfully completed the assignment, she indicates this and,

upon verifying that she really wants to submit her program, it is packaged up and submitted to the server for grading.

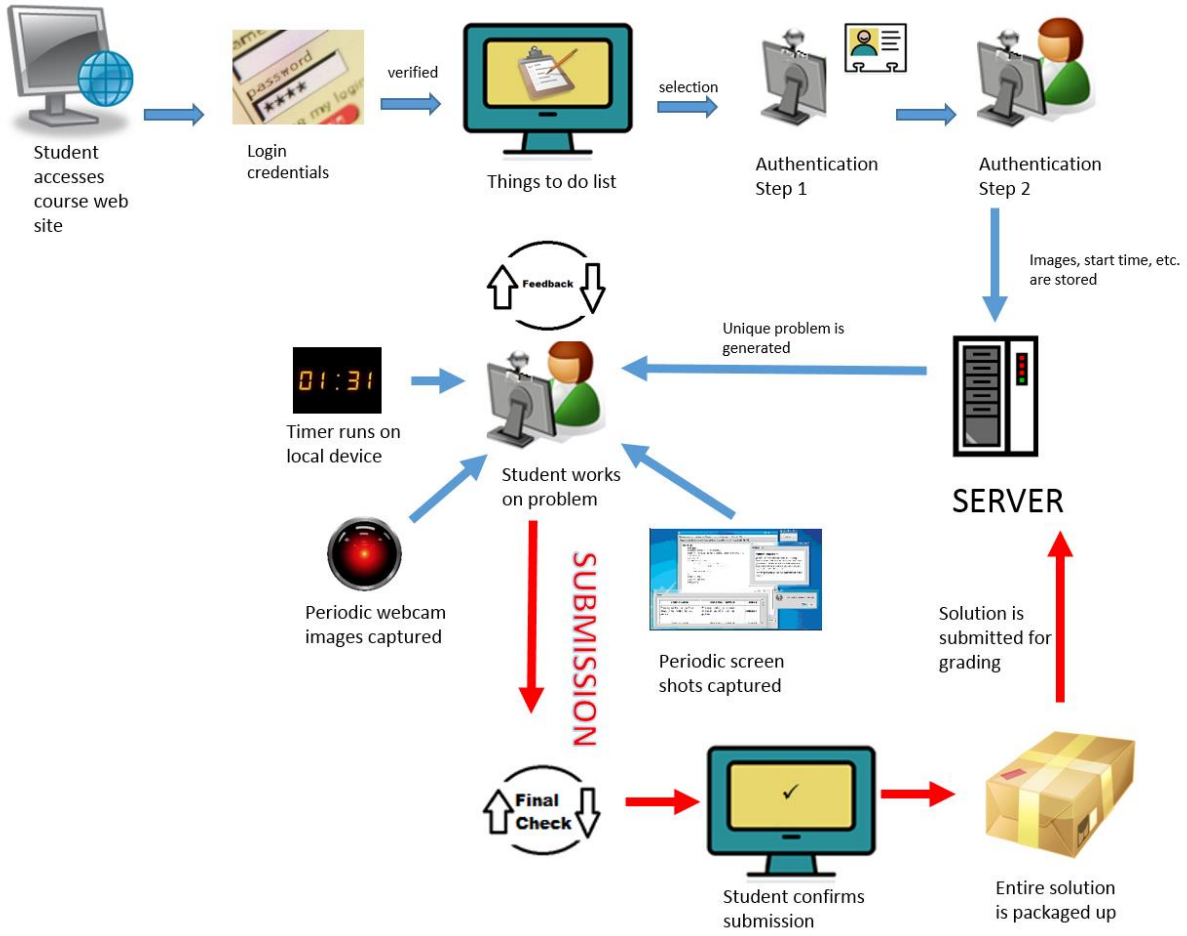

Figure 1. Student user flow.

IX. Evaluation of submissions

When the instructor or TA wishes to grade the submissions, after authentication steps have been executed, the system will present the instructor or TA with the next submission package in the queue. The submission package will contain (1) the user identification information, including the user id and photo, (2) the problem specification, (3) the student's submitted code, (4) results of running the student program. Note that the program must have passed the input/output tests before the program was allowed to be submitted, so at this point the program should at least mostly be correct. Certain types of pathological programs cannot be caught via input/output tests and must undergo a visual inspection to catch them [11]. Thus, the instructor reviews the entire package, looking for anomalies in the code or in the captured screen shots. The student code can be quickly exported

to the instructor's own IDE for testing, in case it's not clear if there is a problem with the program. If any issues are found, the instructor rejects the submission and notes why the program was rejected. Unless a security violation is encountered, the next time the student logs in to the system, the reasons for the rejection are displayed with an instructional prescription, and the student is directed to engage another problem from the library at the appropriate performance level. Experience shows that reviews of acceptable submissions can take less than a minute; submissions with issues that require feedback usually take longer. However, in all cases, the amount of time spent grading is reduced.

X. Conclusions and future work

This paper detailed the findings of over a decade of experience and development of an online assessment system for computer programming students. These findings are being used as specifications on a next generation system. Development of this fourth generation programming assessment system is underway, and it is anticipated that a working demonstration system will be ready for the conference presentation. This version is scheduled for production use in the summer of 2015, and will support multiple online courses. Future potential uses might include integrating the system within a Massive Open Online Course (MOOC).

XI. Bibliography

1. Ambrose, S. A.; Bridges, M. W.; DiPietro, M., Lovett, M. C.; Norman, M. K. (2010) *How Learning Works: Seven Research-Based Principles for Smart Teaching.* Jossey-Bass.
2. Bergadano, F.; Gunetti, D.; and Picardi, C. (2003) Identity verification through dynamic keystroke analysis. *Intelligent Data Analysis, 7 (5)*, 469-496.
3. Gentile, J.R.; Lalley, J.P. (2003) *Standards and Mastery Learning: Aligning Teaching and Assessment So All Children Can Learn*. Corwin Press.
4. Haughton, N.; Yeh, K.-C.; Nworie, J.; Romero, L. (2013) Digital disturbances, disorders, and pathologies: A discussion of some unintended consequences of technology in higher education. *Educational Technology. 53 (4)*, 3-16.
5. Higher Education Opportunity Act (HEOA), 2008. [Online]. Available: http://www2.ed.gov/policy/highered/leg/hea08/index.html
6. Lang, J. M. (2013) *Cheating Lessons: Learning from Academic Dishonesty.* Harvard University Press.
7. Ritter, F. E.; Yeh, K.-C.; Cohen, M. A.; Weyhrauch, P.; Kim, J. W.; Hobbs J. N. (2013) Declarative to procedural tutors: A family of cognitive architecture-based tutors. In Proceedings of the *22nd Conference on Behavior Representation in Modeling and Simulation*. (Centerville, OH, USA, 2013). 108-113
8. Shaffer, S.; Rossen, M. B. (2013) Increasing student success by modifying course delivery based on student submission data. *ACM Inroads 4 (4)*, 81-86.
9. Shaffer, S. (2012) Distance Education Assessment Infrastructure and Process Design Based on International Standard 23988. *Online Journal of Distance Learning Administration 15 (2).*
10. Shaffer, S. (2006) Toward a Systems Dynamics Model of Teaching Computer Programming via Distance Education. Doctoral Thesis. The Pennsylvania State University.
11. Shaffer, S. (2000) Data Partitioning, Code Inspections and Pathological Programs. *Software Quality*.

12. Shaffer, S. (2005). Ludwig: An Online Programming Tutoring and Assessment System. *SIGCSE Bulletin 37 (2)*, 56-60.
13. Sweller, J. (1988) Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12 (2)*, 257-285.