Empirical Investigation throughout the CS Curriculum

David Reed
Department of Mathematics and Computer Science
Dickinson College
Carlisle, PA 17013
reedd@dickinson.edu

Craig Miller
School of CTIS
DePaul University
Chicago, IL 60604
cmiller@cs.depaul.edu

Grant Braught
Department of Mathematics and Computer Science
Dickinson College
Carlisle, PA 17013
braught@dickinson.edu

Abstract

Empirical skills are playing an increasingly important role in the computing profession and our society. But while traditional computer science curricula are effective in teaching software design skills, little attention has been paid to developing empirical investigative skills such as forming testable hypotheses, designing experiments, critiquing their validity, collecting data, explaining results, and drawing conclusions. In this paper, we describe an initiative at Dickinson College that integrates the development of empirical skills throughout the computer science curriculum. At the introductory level, students perform experiments, analyze the results, and discuss their conclusions. In subsequent courses, they develop their skills at designing, conducting and critiquing experiments through incrementally more open-ended assignments. By their senior year, they are capable of forming hypotheses, designing and conducting experiments, and presenting conclusions based on the results.

1 Motivation

Computer science curricula have long emphasized problem solving and theoretical analysis as the central means for learning and reinforcing the discipline's concepts and principles. With this emphasis, however, little attention has been given to skills involving empirical investigation such as forming testable hypotheses, designing experiments, critiquing their validity, collecting data, explaining results, and drawing conclusions. The absence of these skills from computer science curricula is readily apparent by examining the 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force [12]. While this report does identify "abstraction" as an important process and further defines it as including many of the elements found in the experimental sciences, it does not address empirical elements directly. Moreover, it does not offer ways in which these skills can be practiced, nor does it establish competencies expected of students.

A survey of recent SIGCSE proceedings produces only a handful of papers with emphasis on empirical methods. In [9], Schneider stresses experimentation, but in the context of a computational science program that bridges computer science and other academic disciplines. Other recent examples involve the use of experimentation in an upper-level course as a tool for studying complex systems ([4] and [8] in operating systems; [3] in human-computer interaction; and [14] in scientific programming). At the introductory and intermediate levels, however, there is almost no coverage of these topics. While textbooks generally provide principles for designing and implementing computer systems, there is rarely any guidance on empirically evaluating or investigating their properties once they are created. The texts rarely provide testing methods, and when they do, it is often a test for whether the system works as opposed to a performance comparison to alternate models. When comparisons between models are made, it is typically only when theoretical analysis is possible and therefore it is the theory that is taught.

Despite the dearth of coverage, empirical skills are playing an increasingly important role in the computing profession and our society. If we wish our graduates to be capable experimenters and evaluators of experimental data, then a more systematic coverage of empirical methods throughout the curriculum is needed. Just as we do not expect our students to be experts at program design and testing until they have had years of experience with the design process, we should expect no more of them when it comes to designing and evaluating experiments.

2 Benefits

While we expect problem solving and theory to remain the cornerstone of computer science, the discipline does present a rich set of phenomena open to empirical investigation. Moreover, the student who learns and practices these skills has much to gain from them. In particular, they benefit the student as a future computing professional, as a consumer of scientific knowledge, and, most immediately, as a learner of the discipline's content.

The practice of empirical investigation prepares a student for a career in computing, where claims based on benchmarks, test suites and perceived usability are common. Unfortunately, the evidence for these claims is not always well grounded, and the traditional computer science curriculum does little to prepare a student for successfully critiquing their validity. Training in experimental methodology exposes students to potential flaws in conclusions based on empirical evidence and teaches how to compensate for them. In the future, empirical methods will be essential to evaluating and validating hardware and software systems as they become increasingly complex and defy the kinds of assumptions needed for theoretical analysis. In the wake of the increasing importance of empirical investigation, computing professionals have called for more training in experimentation [11,15].

Practicing experimental methodology in the context of a student's expertise provides the best learning situation since students can perform "sanity checks" on conclusions based on their prior knowledge. However, these skills extend beyond the computer profession as many scientific, economic and social claims are based on empirical studies. As such, a practiced consumer of empirical evidence may apply it to one's health, finance, and citizenship. While computer science students may receive such training in other disciplines, it typically will not be a systematic experience lasting several courses and it will not be in the context of their expertise.

As a further benefit of performing empirical investigations, students gain practice presenting and explaining results. Research in learning and cognition suggests that the use of investigation and explanation, as opposed to problem-solving, often results in a better understanding of a discipline's content. For example, a suite of studies identifies the practice of technical and scientific explanation [2] as the essential element to successfully learning scientific principles and concepts. Other studies report that the overly specific nature of problem solving under an engineering model is not often the best way to acquire an understanding of the problem's domain [10,13,5] (see [6] for a review). In these cases, a less task-specific goal consistent with scientific hypothesis formation and testing is more appropriate for learning a system's principles. While formal mathematical analysis typically addresses these concerns, it is not the appropriate tool for many emerging areas of computer science, which include software usability, programming methodology, and their evaluation.

3 Developing Empirical Skills

There are many parallels between programming skills and empirical skills. It is generally accepted that developing expert skills as a program designer and evaluator takes years of experience. In most curricula, skills are developed gradually (see [1,7] for rationale). At the introductory level, students do very little design on their own. Instead, they learn by studying and modifying existing programs. While they may contribute to simple designs, it is usually in the context of a well-defined infrastructure (e.g., class interfaces). At the intermediate level, students are expected to be able to design small programs or to manage more complex systems with minimal infrastructure. Finally, after this cumulative experience, they become capable designers and can be expected to use their skills to solve problems in upper-level courses.

This same gradual process can be applied to developing empirical skills across the CS curriculum. Early on, students will learn by performing experiments, analyzing the results, and perhaps most importantly, discussing their conclusions. By the end of the process, they should be capable of forming testable hypotheses, designing and conducting experiments, and presenting conclusions based on the results.

3.1 Introductory Level

As was the case with program design, it is somewhat unreasonable to expect students to design good experiments at the introductory level. Instead, the focus should be on exposing them to experimental methods, and teaching them to use experiments to study and analyze systems. Through experimentation, students are able to study and solve interesting problems even before they have developed programming skills. In addition, the foundation is laid for further empirical skills to be developed later.

For example, the very first lab in our intro course involves experimentation with random sequences of letters. The students are given code for generating and displaying random N-letter sequences of letters (where N can be changed). Using this program, they are led through the process of trying to approximate how many 3-letter words there are in the English language. This is done by having them generate a large number of 3-letter sequences and count the number of words that occur. (A side issue that they are asked to explore is just how many sequences must be generated to obtain a reasonable result.) Once they have obtained a number, they can use it to estimate the relative probability of obtaining a word from a random letter sequence. For example, if they generate 15 words out of 500 random 3-letter sequences, then they may hypothesize that 3% of all random 3-letter sequences are words. Since the total number of 3-letter sequences is easy to compute ($26^3 = 17,576$), they may then estimate the total number of 3-letter words ($17,576 * .03 = 527$). Once they have been led through this experiment, they are asked to repeat it in order to estimate the number of 4-letter words.

Random events such as coin flips and dice rolls provide many opportunities for simulation and experimentation in an introductory course. In addition to giving students practice with programming constructs such as loops and counters, experimentation with random events can help to remove many misconceptions students may have regarding probabilities. For example, an early exercise asks the following question: *Suppose you roll two 6-sided dice 1000 times. Which would you expect to get more of, 7's or 2's?* Many students respond that this is impossible to predict since each roll of the dice is a random event. Actually performing repeated experiments, simulating the dice rolls and keeping counts of the totals, is far more effective in dispelling this misconception than lectures.

At the introductory level, students can also be exposed to empirical methods as a means of studying complex systems. Towards the middle of our intro course, students experiment with two different forms of random walk. They are given code that simulates a 1-dimensional random walk. The analogy is that of a person standing in the middle of an alley. At the start, she is N steps away from either exit, and each step that she takes is in a random direction towards one of the exits. The students are then led through a series of experiments to try to determine how many steps it takes (on average) to exit the alley. Once they have accomplished this, they are asked to consider a slight variation of this random walk, where the person is standing at the end of a dead-end alley, whose exit is N steps away. They are asked to hypothesize as to whether this modified random walk should take more or fewer steps (on average) than the previous type. They then must modify the code and use it to test their hypothesis. A considerable portion of the grade for this lab is based on the explanation of the results of their experiment and how it verifies or refutes their hypothesis.

Finally, experimentation can be utilized at the introductory level to better the students' understanding of efficiency. The basic idea of rate-of-growth analysis can be exemplified by having students execute sorting algorithms on increasingly large data sets. That is, they can verify experimentally that doubling the size of the input set causes an $O(n^2)$ sort to take four times as long, while an $O(n \log n)$ sort takes only a little more than twice as long. This point can be further emphasized by providing a mystery sorting function, whose details are hidden from the student. Simply by performing experiments, the student should be able to hypothesize as to which complexity class that algorithm belongs. More advanced questions can also be posed for experimentation, such as which algorithms are most sensitive to pre-existent data ordering or the size of the data being sorted.

3.2 Intermediate Level

Having been exposed to experimental methods at the introductory level, students require less infrastructure and are even able to design and conduct small experiments on their own. In addition to more complex simulations of the type described above, there are numerous opportunities for empirical studies at the intermediate level of the curriculum. Experimentation can be used to help characterize the efficiency of data structures and algorithms, as well as compare and contrast different implementations.

For example, a natural application for studying the behavior of queues is with a bank simulation. Having experienced them in real life, students are able to describe some of the tradeoffs between multiple-line (one per teller) and single-line (served by multiple tellers) models. Students usually recognize that a single-line model is fairer, since customers are served in the order in which they enter the bank. On the other hand, it is possible for a particular customer to be served more quickly in a multiple-line model by selecting the fastest line. Using a random number generator to simulate the arrival and transaction lengths of customers, their intuitions can be verified or refuted by experimentation. Complex connections between variables such as the customer arrival rate, maximum transaction time, number of lines, and number of tellers can also be explored.

As more complex data structures are introduced for solving problems, their properties can be verified experimentally in addition to (or instead of) theoretically. For example, when covering binary search trees in our data structures course, we consider the costs and tradeoffs involved with keeping the tree balanced. The students are shown a theoretical result, that randomly inserting N values into a binary search tree will result in a tree whose maximum depth is roughly

(2 log N). For an assignment, they must then verify this result experimentally. While the idea for the experiment is provided for them, the details of its design and implementation are left to the students.

Similarly, experimentation can be used to compare and contrast alternate implementations of data structures. For example, various strategies for handling collisions in a hash table (e.g., chaining, linear probing, quadratic probing) can be presented and compared by timing each implementation on random data sets. In another example, students study the efficiency of a dictionary, using different implementations (e.g., binary search tree, hash table, trie).
These same techniques for comparing and contrasting data structures can be applied to algorithms. Experimentation can be used to verify the complexity of competing algorithms, such as Floyd's algorithm and Dijkstra's algorithm for finding shortest paths. In addition, many of these algorithms have theoretical performance predictions that only hold under certain assumptions. For example, a theoretical prediction for hashing with linear probing assumes that each key is equally likely. In practice, however, the most frequently used keys may correspond to elements that were first inserted into the table. In this case, experimental results would show that the average number of collisions is less than the theoretical prediction. At this level, we may expect a student to write a concise paragraph describing his or her results and explaining why they differ from the theoretical predictions.

Experimentation and explanation also play an important role in our Computer Organization class, where the students design, build and ultimately program a virtual machine emulator. In addition to implementing components such as adders, ALUs, latches, and register banks, they must design tests that verify the correctness of their solutions. Grades are not only based on the correctness of the solutions but also on the completeness of the tests and the presentation of the results. Other examples might include implementing and comparing the performance of carry ripple vs. carry look-ahead adders, micro-programmed vs. hardwired control units, and various cache line replacement policies.

3.3 Upper Level

Having developed their experimental design skills through incrementally more open-ended experiments, students at the upper level of the curriculum are better prepared to design experiments of their own. They should also be capable of justifying their designs and how their results support or refute initial hypotheses. At Dickinson College, a project in an advanced course typically requires a short report that presents the experimental design, describes the results, and explains their consequences. Ideally, it follows the format of a scientific research paper.

Systems courses afford many opportunities for empirical inquiry. In a database systems course, students are assigned a project to compare the retrieval efficiency of a clustered index relative to an unclustered index. The target entries of a clustered index are generally on the same pages whereas unclustered entries are scattered across many pages. Thus, the students' experiments should reveal that fewer read operations are required for the clustered index whenever multiple entries are retrieved. The project assignment states the empirical question and specifies the independent and dependent variables. The students were then required to write code for collecting the results and submit a one-page single-spaced report that described and explained them.

Another example of a course that assumes and makes extensive use of empirical skills is our Microcomputer Interfacing course. This course explores, in a laboratory setting, ways to interface electronics, actuators and sensors to a computer for control and data acquisition purposes. Throughout the semester students calculate the theoretical performance of systems, design experiments to measure performance and compare their results to the theory. As one example, in the lab on A/D converters students are given the task of calculating and experimentally verifying the resolution of an A/D converter. The theory behind the calculations is covered in class, however the experiment is designed entirely by students working in groups of two. Other assignments, such as designing a temperature control system for a warehouse, are designed to force the students to make design decisions. As they do so, they are asked to explain the tradeoffs involved in their design, such as why they implemented parts of the system in hardware as opposed to software and vice-versa. When there is not a clear-cut way of making a design decision, several implementations can be sketched out and compared theoretically or implemented and compared empirically. Evaluation of each lab is based on a lab notebook kept in a format similar to what might be used in a research laboratory. The grade for the lab portion of the course is based on how well the students design, conduct, document and explain their experiments in the notebooks.

Whether integrated into the curriculum or offered as a separate course, the study of human-computer interaction is playing an increasingly important role in computer science programs. Perhaps more than any other area, mastery of empirical investigative skills is critical for its successful study. Students who have already practiced experiment design and implementation will have a needed start towards addressing the additional complexities of human behavior. At this level, students can be expected to formulate testable usability objectives, experimentally verify them, and justify prescribed improvements to the user interface.

The process of empirical skill development culminates in the senior seminar, where the students are expected to design, implement, and analyze experiments in the context of a capstone project. As an example, one student developed a greedy algorithm for assigning students to freshman seminars and compared its performance to the ad-hoc method used by the college. Other students have compared the performance of the Solaris OS running on different hardware platforms, evaluated a genetic scheduling algorithm for Linux, and measured the capacity of a Linux web-server for serving text, graphic and CGI generated web pages. Each project requires an oral presentation and written paper, as actually practiced in scientific communities.

4 Conclusion

Experience suggests that first-year students rarely understand what constitutes a good experiment. Furthermore, most have not developed the skills necessary to design and conduct quality experiments. The initiative described in this paper addresses these deficiencies using an incremental approach to teaching and practicing empirical investigative skills. By their senior year, our students are capable of forming hypotheses, designing and conducting experiments, and interpreting the results, as demonstrated in their capstone projects. In addition, the emphasis on explaining empirical phenomena throughout the curriculum requires students to link concrete results to abstract concepts, improving their understanding of both empirical investigation and the underlying core computer science concepts.

References

[1] Astrachan, O., and D. Reed (1995). “AAA and CS1: The applied apprenticeship approach to CS1.” *SIGCSE Bulletin* **27**(1): 1-5.

[2] Chi, M.T.H., M. Bassock, M.W. Lewis, P. Reimann, and R. Glaser (1989). “Self-explanations: How students study and learn examples in learning to solve problems.” *Cognitive Science* **13**: 145-182.

[3] Clarke, M.C. (1998). “Teaching the empirical approach to designing human-computer interaction via an experimental group project.” *SIGCSE Bulletin* **30**(1): 198-201.

[4] Downey, A.B. (1999). “Teaching experimental design in an operating systems class.” *SIGCSE Bulletin* **31**(1): 316-320.

[5] Miller, C., J. Lehman, and K. Koedinger (1999). “Goals and learning in microworlds.” *Cognitive Science* **23**(3).

[6] Nhouyvanisvong, A., and K. Koedinger (1998). “Goal specificity and learning: Reinterpretation of the data and cognitive theory.” *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, Erlbaum**:** 764-769.

[7] Pattis, R.E. (1991). “A philosophy and example of CS1 programming projects.” *SIGCSE Bulletin* **23**(1): 34-39.

[8] Robbins, S., and K.A. Robbins (1999). “Empirical exploration in undergraduate operating systems.” *SIGCSE Bulletin* **31**(1): 311-315.

[9] Schneider, G.M. (1999). "Computational science as an interdisciplinary bridge.” *SIGCSE Bulletin* **31**(1): 141-145.

[10] Sweller, J. (1998). “Cognitive load during problem solving: Effects on learning.” *Cognitive Science* **12**: 257-285.

[11] Tichy, W.F. (1998). “Should computer scientists experiment more?” *Computer* **31**(5): 32-40.

[12] Tucker, A.B., editor (1992). "Report of the ACM/IEEE-CS Joint Curriculum Task Force.” The Association for Computing Machinery, New York.

[13] Vollmeyer, R., B. Burns, and K. Holyoak (1996). “The impact of goal specificity on strategy use and the acquisition of problem structure.” *Cognitive Science* **20**: 75-100.

[14] Zachary, J.L. (1997). “The gestalt of scientific programming: Problem, model, method, implementation, assessment.” *SIGCSE Bulletin* **29**(1): 238-242.

[15] Zelkowitz, M.V., and D.R. Wallace (1998). “Experimental models for validating technology." *Computer* **31**(5): 23-31.