Core Empirical Concepts and Skills for Computer Science

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

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

David Reed
Department of Mathematics and Computer Science
Creighton University
Omaha, NE 68178
davereed@creighton.edu

ABSTRACT
Educators are increasingly acknowledging that practical problems in computer science demand basic competencies in experimentation and data analysis. However, little effort has been made towards explicitly identifying those empirical concepts and skills needed by computer scientists, nor in developing methods of integrating those concepts and skills into CS curricula. In this paper, we identify a core list of empirical competencies and motivate them based on established courses outside of computer science, their potential use in standard CS courses, and their application to real-world problems. Sample assignments that facilitate the integration of these competencies into the CS curriculum are also discussed.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum*. G.3 [**Probability and Statistics**]. A.0 [**General**]: Conference Proceedings.

General Terms
Experimentation.

Keywords
Empirical concepts, empirical skills.

1. INTRODUCTION
The questions faced by computer scientists are often empirical in nature, requiring more than just theoretical analysis. For example, consider optimizing an enterprise server, assessing a new software development methodology, comparing the latest microprocessors, designing a user interface, or investigating natural phenomena via simulation. To be meaningful, the results of these and similar tasks must rely on data collection and analysis from experiments conducted under typical or "real-world" conditions. While useful in many applications, theoretical analysis is not always feasible, particularly when variable or stochastic factors come into play. For example, optimizing server performance requires first identifying the conditions and workload under which the system is expected to function, and then testing the system under those conditions. Likewise, assessing a new method for developing software requires analyzing factors such as management structure and the variability of human performance. In short, many of the most meaningful real-world claims in computer science critically depend on the evaluator's ability to conduct valid experiments and analyze their results.

The importance of experimentation and empirical reasoning in computer science has long been recognized, as illustrated by Newell and Simon's 1975 Turing Award lecture entitled *Computer Science as Empirical Inquiry: Symbols and Search* [13]. More recently, Computing Curricula 2001 acknowledged the importance of empirical methods with the following statement:

> *"The scientific method represents a basis methodology for much of the discipline of computer science, and students should have a solid exposure to this methodology."* [10]

While Computer Curricula 2001 does suggest that "students may acquire their scientific perspective in a variety of domains", it does not provide guidance as to which empirical concepts are important or how they can be integrated into the computer science curriculum. Traditionally, computer science curricula have tended to emphasize problem solving and theoretical analysis as the central means for learning and reinforcing the discipline's concepts and principles [14,16]. Empirical skills, while expected of computer science students by graduation, have received little attention in computer science programs. In the words of Paul Schneck:

> *"Many (most?) students of computer science are not educated as scientists. They are trained as programmers. This results in a situation, reflected in our literature, where many practitioners form unstructured phenomenological inferences instead of creating models, forming hypotheses, and performing experiments to validate (or invalidate) the hypotheses and models."* [12]

In recent years, several articles have appeared in the literature advocating the use of experimentation in computer science courses. However, these articles tend to involve applying empirical reasoning to a problem in a particular course, such as operating systems [7,15], organization/architecture [1,4,17], software engineering [3,9], or human-computer interaction [6]. While these examples demonstrate the utility of experimentation, the broader issue that must be addressed is the need for a systematic approach to developing empirical skills across the computer science curriculum. Similar to the way that programming and problem-solving skills are developed incrementally, important empirical concepts and skills must be introduced early and revisited often in order to be mastered by students.

As a first step towards developing a systematic approach to empirical reasoning, this paper proposes a list of competencies

that should be expected of all graduates in computer science. These competencies include those empirical concepts that must be understood by students in order to appreciate important aspects of computing, as well as specific empirical skills that are required for experimental studies within the field. We motivate the need for these concepts and skills by their appearance in standard Quantitative Reasoning and statistics courses, as well as their application to practical real-world applications in computer science. A few specific assignments are also reviewed to illustrate how the core empirical competencies can be integrated into a traditional computer science curriculum.

2. CORE EMPIRICAL COMPETENCIES

Table 1 presents our candidate list of core empirical competencies for practicing computer science. In constructing this table, we first examined typical tasks in which computer scientists apply empirical reasoning, and extracted those concepts and skills necessary for the tasks. We also reviewed the contents of numerous Quantitative Reasoning and basic statistics courses [8], and met with colleagues both in computer science and statistics. We believe the content of this table provides a framework for successfully integrating empirical reasoning into computer science curricula.

We have broken the list into three different levels corresponding to the levels of the undergraduate computer science curriculum. The purpose of this division is twofold. First, it emphasizes an incremental approach to developing empirical knowledge and skills. Generally, the concepts and skills appearing at the introductory level are basic and less formal, those appearing at more advanced levels require a more formal understanding and pair well with upper division computer science courses. For example, at the introductory level, students learn that a larger sample is more likely to produce a more accurate estimate of the population but do not learn about confidence intervals for quantifying its accuracy until the advanced level. This organization allows for a progressive introduction of many empirical concepts, encouraging students to develop intuition about them before learning more formal or mathematical definitions.

The second reason for dividing the competencies into levels is to complement the traditional format of computer science curricula. In most undergraduate programs, introductory-level courses focus on programming and problem-solving skills. Intermediate-level courses (e.g., data structures, algorithms, computer organization) emphasize foundational knowledge and skills, introducing more formality in design and analysis. Finally, upper-level courses build upon these foundations to study specific areas of computing in depth.

Table 1 organizes the empirical concepts and skills to best fit with the traditional practices in curriculum design just described. For example, many of the concepts listed at the introductory level could easily be introduced in typical CS1 and CS2 assignments, such as simulating dice rolls or random walks. Basic data representation using scatter-plots appears at the intermediate level because graphing problem size versus running time is a natural thing to do in a data structures or algorithms course. It is important to note that we do not view the divisions that appear in Table 1 as absolute or inflexible. One instructor might take a more formal approach to algorithms in CS1 and CS2, introducing standard deviation or curve fitting early. Likewise, another instructor might choose to defer other topics to later in the curriculum.

Table 1: Core empirical competencies for computer science

Level	Concepts	Skills
Introductory	• mean vs. median • informal definition of probability • chance error • expected values • Law of Large Numbers • consistency vs. accuracy • benefits/limitations of models	• Be able to conduct a well-defined experiment, summarize the results, and compare with the expected results. • Be able to evaluate the persuasiveness of experimental conclusions, focusing on issues such as the clarity of the hypothesis, the sample size, and data consistency.
Intermediate	• uniform vs. normal distribution • standard deviation • sampling large populations • methods for obtaining a good sample • curve-fitting (e.g., linear regression) • data presentation (e.g., tables, scatter-plots, histograms) • software testing & debugging strategies • validity threats (e.g., confounding variables, non-generalizability)	• Be able to plot and describe the relation between two variables, such as problem size vs. efficiency when analyzing algorithms. • Be able to use a sample to make predictions about a population. • Be able to evaluate the persuasiveness of experimental conclusions, focusing on issues such as appropriate statistical measures, relevance of a model, and generality of the conclusions. • Be able to design a test suite for a software project and conduct systematic debugging.
Advanced	• standard error • confidence intervals • measures of goodness of fit (e.g., correlation coefficient) • observational study vs. controlled experiment • correlation vs. causality • significance tests (e.g., t-test, z-score)	• Be able to apply experimental methods to the analysis of complex systems in different domains. • Be able to formulate testable hypotheses and design experiments for supporting or refuting those hypotheses.

The following sections look at each level of competencies, providing justification for the concepts and skills listed. At each level, sample assignments that can be used to introduce and motivate particular competencies are also described.

2.1. Introductory-level Competencies

The main empirical goals at the introductory level are to instill a basic understanding that programs are used to solve problems, and to provide informal tools for experiencing this process. Learning to program is typically difficult enough for beginning students, so the amount of class time that can be devoted to additional empirical concepts is minimal. Instead, we focus on the types of activities that complement traditional programming assignments, but with an empirical flavor. For example, students would not be expected to design an experiment at the introductory level, but they could be given an existing experimental design (say in the form of a simulation program) and be asked to carry out the experiment and analyze the results. A simple example would be to have a student write a short program that simulates two dice, then execute that program and verify the expected distribution of dice totals. Or, the student might be asked to simulate different variations of random walks and verify or refute hypotheses about expected distances of those walks. Assignments such as these involve standard programming constructs (e.g., conditionals, loops, counters) but also demonstrate that the problem is not solved as soon as the program compiles. Instead, students must collect data by executing the program repeatedly, and analyze the results.

The empirical concepts listed at the introductory level support the types of activities described above. Most involve descriptive statistics, such as mean and median. These concepts provide informal tools for analysis, allowing students to begin the process of empirical reasoning in an intuitive manner. Assignments that involve writing Monte Carlo simulations, such as dice rolls and random walks, require students to perform analysis of their program's output and draw conclusions, providing an opportunity to initiate an elementary discussion of probability, consistency, accuracy and expected values. In addition, students discover that unexpected results may be due to their programs correctly simulating an incorrect model of the process, or to bugs in their program [2]. They learn that larger samples tend to produce more consistent results and, assuming a correct program, more accurate results.

2.2. Intermediate-level Competencies

The intermediate level provides a transition from following prescribed experiments to successfully designing experiments. While informal measures may have sufficed at the introductory level, problems such as selecting the best sorting algorithm for a particular application, optimizing the hashing scheme for a large hash table, or selecting the appropriate page size in a virtual memory system require more formal analysis tools. More formal statistical concepts are introduced, including quantitative measures describing distributions, lines and curves. For example, students may learn the formal distinction between uniformly and normally distributed random variables. While at the introductory level, the normal distribution may have been understood in terms of summing dice and learning that "values in the middle are more probable," the intermediate level may show probability distributions and use concepts from calculus. At this point, students may develop mathematical models and estimate parameters to fit a line, curve or normal distribution to experimental data and then use the fit to make predictions.

Since the intermediate level is typically where students begin designing and implementing complex software systems, it is the natural place to emphasize the connection between software testing/debugging and experimentation. When testing a system involving many interacting components, a methodical approach to designing test suites and identifying bugs is necessary. For example, debugging a program might involve observing its behavior on a variety of inputs, forming a hypothesis as to what might be going wrong (and where the error is in the code), and then executing on additional input sets to test that hypothesis. Empirical skills developed at the introductory level can naturally be applied here.

We also suggest that students start critiquing experimental design at the intermediate level. Drawing from experimental methodology courses in psychology, a systematic critique of experiments comes from reviewing threats to an experiment's validity. These threats include problems with cause and effect, statistical validity, and whether the results generalize to real-world situations. A critique of cause-and-effect checks whether the experiment successfully manipulates the factor being tested while controlling all other variables. Checking for statistical validity involves verifying that the sample is large enough to draw conclusions. Finally, students should learn to assess whether the circumstances and assumptions in the experiment are close-enough to useful real-world situations for the result to apply to them.

2.3. Advanced-level Competencies

At the advanced level, students are expected to be able to design, conduct, and analyze experiments to address problems in various settings. Building upon the foundational knowledge and skills developed in earlier courses, a student in a human-computer interaction course should be able to evaluate different user-interfaces empirically, using proper statistical measures and controlling for extraneous variables. Likewise, a student in a databases class should be able compare different indexing strategies using experimentation.

The specific concepts introduced at the advanced level will vary depending upon the course and the application-specific knowledge required to conduct experiments in that domain. For example, studies involving human behavior might require learning different techniques for assigning human participants to an experiment's conditions. Depending on the course, students might learn about the different kinds of studies, such as the difference between a controlled experiment and an observational study. Some advanced statistical measures, such as confidence intervals, correlation coefficients, and p-values, will be useful in a variety of domains, and so are listed in Table 1.

Having developed empirical reasoning skills throughout the curriculum, students at the advanced level are prepared to rigorously compare theoretical predictions with empirical results and to discuss the strengths and weaknesses of both approaches. Calculating confidence intervals for the empirical results helps the students assess whether the two approaches are consistent with each other for a given application. Particularly large confidence intervals expose limitations of empirical findings whereas inappropriate assumptions may

limit the validity of theoretical predictions. Often discrepancies between the two approaches provide learning opportunities since they motivate students to suggest possible problems with either approach. [11]

3. REAL-WORLD APPLICATIONS

So far we have motivated our list of concepts and skills through their origin in statistics and Quantitative Reasoning courses. We have also discussed how these competencies can be applied to traditional concepts and exercises in computer science curricula. A third way of motivating our selection of competencies is by witnessing their application to real-world problems. In this section we will discuss two such applications. They are the optimization of computer systems (both software and hardware) and the evaluation of benchmark results.

3.1. System Optimization

Very often, opportunities for optimization arise as a computer system is being developed. As prototypical examples of system optimization, we might consider the development of a software class library and the configuration of an enterprise information server. Because these are real systems with myriad implementation details, theoretical analysis is impractical for optimization, thus necessitating the use of empirical studies.

First, consider the task of optimizing a sorting algorithm. While it is true that an O(n log n) sorting algorithm will be faster than an $O(n^2)$ algorithm when the list size is large, an $O(n^2)$ algorithm can be faster on small lists. To optimize an O(n log n) sort such as quick sort or merge sort, one can modify it to revert to the use of an $O(n^2)$ algorithm, such as insertion sort, when the list size falls below some threshold. In fact, this has been done in the Java SDK 1.4.1, The `java.util.Arrays` class provides methods for sorting both primitive and object types utilizing quick sort and merge sort, respectively. Both of these sorts have been optimized by reverting to the use of insertion sort for base-case lists of less than seven elements. In a separate optimization example, tuning an enterprise server involves minimizing the response time of the system while at the same time ensuring that it scales reasonably with both user load and database size. Each of these optimizations requires empirical investigation and motivates several of the empirical skills and concepts on our list.

> **Experiment Design:** Both the sorting optimization and the tuning of an enterprise server require the use of controlled experiments. In the case of the sort optimization, in order to determine the proper threshold at which to revert to insertion sort, the experimenter must be aware of how the large population of input datasets is sampled and the effect of this sampling on the generalizability of the results. In the case of the enterprise server, an experiment might control the load on the server while measuring the response time of the system. In such an experiment, the experimenter must be careful to isolate the variables of server load from database size and content.
>
> **Data Analysis:** For the sort optimization, the concepts of consistency and accuracy can be used to determine when a sufficient number of trials have been performed. This can be further formalized by the use of standard deviation. Finally, curve fitting and linear regression can be used to determine the optimal value at which to switch from the quick sort (or merge sort) to insertion sort. In the case of tuning an enterprise server the analysis of experimental result may be less formal. For example, poor performance with respect to the size or content of the database may prescribe the creation of new database indices.

3.2. Benchmark Evaluation

Benchmarks are commonly used to compare the performance of software and hardware systems. For example, many claims have been made regarding the performance of the PowerPC processor compared to the x86 series of processors. Each new generation of these microprocessors invites new studies comparing their power (see [5] for a summary). Often, claims of superior performance are backed by running a suite of standard benchmarks and comparing average times. Our suggested core competencies provide the concepts and skills needed for understanding and critiquing these studies:

> **Confounding variables.** Sometimes the studies do not succeed in controlling all factors except for the power of the microprocessor. For example, different microprocessors may require different compilers. If one compiler is better engineered, a subsequent difference in performance may be the result of the compilers and not the microprocessors.
>
> **Generalizing results.** A difference in performance among the benchmark tasks does not necessarily mean that a difference in performance will be noticed when conducting real-world tasks. For example, if the benchmark tasks demand a substantial number of floating-point operations but the applied tasks do not, the microprocessor with fast floating-point operations will not show the same advantage when performing the applied tasks.
>
> **Significance tests.** Unless the execution of the benchmarks is carefully controlled to allow no random variation (e.g. from network delays, imprecise clock ticks, processor interrupts, etc.), a large number of executions may be required in order to rule out the effect of noise. Confidence intervals and statistical hypothesis testing may be used.

Of course studying these threats to validity does not guarantee that students will be able to successfully critique or design experiments. However, they do provide students with a language and a systematic process for understanding how experiments might fail. In the case of statistical validity, they also learn general-purpose tools for testing for reliable results.

4. ASSIGNMENT REPOSITORY

As part of an NSF-sponsored project we have begun using our list as a framework for materials development, creating an online repository of assignments that support the core empirical competencies that we have identified. The goal of this repository is to provide instructors with assignments that could be integrated into existing courses with a minimal amount of effort, and yet introduce and exercise empirical skills. For example, at the introductory level, an instructor might select from a variety of assignments that involve Monte Carlo simulations and require the student to collect and

analyze data (using concepts such as mean, median, consistency, and accuracy). An instructor in an algorithms course might select from a variety of topics, such as a comparison of balanced vs, unbalanced tree representations in a dictionary application.

The assignments in the online repository have been classroom tested, many at multiple institutions, to ensure their effectiveness, and new assignments are continually added. In fact, the submission of new assignments or assignment ideas is always welcome. The repository may be accessed at:

http://XXXXXXXXXXXXXX

5. CONCLUSION

We have proposed and motivated a list of core empirical concepts and skills that we believe to be essential to the well-versed computer science graduate. While each of the concepts and skills in our list is motivated by computer science applications, there is certainly some room for debate over the inclusion or exclusion of topics. It is hoped that this paper will serve to open a dialog about the content of such a list, leading to a refinement of the skills and concepts that it contains. In a broader context we hope that this dialog will raise awareness of the need to address empirical skills and concepts in the computer science curriculum. In addition, a concrete list represents a first step toward being able to assess whether students are really learning competencies needed for solving practical problems using experimentation. Finally, the list provides a framework to guide the development of materials (such as the assignments found at our online repository) that reinforce computer science concepts while introducing and exercising empirical skills.

6. ACKNOWLEDGEMENTS

Partial support for this work was provided by the National Science Foundation's Course, Curriculum, and Laboratory Improvement Program under grant DUE-0230950. We would also like to thank Mike Fries, Mike Kenniston, and David Jabon of DePaul University, who met with the authors and provided valuable feedback on the competencies list.

7. REFERENCES

[1] Braught, G., and D. Reed (2001). "The knob & switch computer: a computer architecture simulator for introductory computer science." *ACM Journal of Educational Resources in Computing* **1**(4).

[2] Braught, G., and D. Reed (2002). "Disequilibration for Teaching the Scientific Method in Computer Science" *ACM SIGCSE Bulletin* **34**(1): 106-110.

[3] Basili, V.R. (1996). "The role of experimentation in software engineering: past, current, and future." *IEEE Proceedings of ICSE 18*.

[4] Bem, E.Z. (2002). "Experiment-based project in undergraduate computer architecture." *ACM SIGCSE Bulletin* **34**(1): 171-175.

[5] Blatchford, N. (2003). "Analysis: x86 vs. PPC." *OS News*. Retrieved August 4, 2003 from http://www.osnews.com/story.php?news_id=3997.

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

[7] Downey, A.B. (1999). "Teaching experimental design in an operating systems class." *SIGCSE Bull*etin **31**(1): 316-320.

[8] Freedman, D., Pisani, R., and Purves, R. (1998). *Statistics, Third Edition*. New York: W.W. Norton & Company.

[9] Hendrix, T.D., J.H.Cross II, S. Maghsoodloo, and M.L.McKinney. (2000). "Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java." *ACM SIGCSE Bulletin* **32**(1): 382-386.

[10] Joint IEEE Computer Society/ACM Task Force for CC2001 (2001). "Computing Curricula 2001." Online at http://www.acm.org/sigcse/cc2001/.

[11] Miller, C.S. (2003). "Relating Theory to Actual Results in Computer Science and Human-Computer Interaction. *Computer Science Education* **13**(3): 227-240.

[12] Mudge, T. (1996). "Report on the panel: How can computer architecture researchers avoid becoming a society for irreproducible results?" *Computer Architecture News* **24**(1): 1-5.

[13] Newell, A., and H.A.. Simon (1976). "Computer Science as empirical inquiry: symbols and search (ACM Turing Award Lecture)." *Communications of the ACM* 19, 111-126.

[14] Reed, D., Miller, C., and Braught, G. (2000). "Empirical Investigation throughout the CS Curriculum." *ACM SIGCSE Bulletin* **32**(1): 202-206.

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

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

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