Performance Lies My Professor Told Me:

The Case for Teaching Software Performance Engineering to Undergraduates

Robert F. Dugan Jr.
Department of Computer Science
Stonehill College
Easton, MA 02357, U.S.A
bdugan@stonehill.edu

ABSTRACT

In this paper we report a survey examining the approach to performance and software engineering in courses at highly ranked computer science schools in the United States. An analysis of the survey shows serious shortcomings including inadequate or missing definitions of performance, reactive "fix it later" mentality, vague performance requirements, and a general lack of awareness of the practices developed by the Software Performance Engineering (SPE) community. The survey is followed by guidelines for teaching SPE to undergraduates based on a semester long course we have developed. It is our plan to incorporate these guidelines into the curriculum of our senior capstone software engineering course.

Categories and Subject Descriptors

H.4.m [**Software Performance Engineering**]:

Keywords

Software Engineering, Performance, Education

1. INTRODUCTION

The Software Performance Engineering (SPE) community exists, in part, because of endless problems that industry experiences with application performance. Smith's classic paper addresses the root cause of these problems by calling for a proactive approach to performance during the software development cycle which she coined "Software Performance Engineering" [23].

Despite the excellent work of SPE researchers and practitioners alike, many in the community would agree that software development still has a costly "fix it later" performance methodology. One source of this methodology may be the educational system producing the workforce that creates software applications. To investigate this hypothesis, we conducted a survey of undergraduate computer science courses that included software engineering and performance.

After reviewing courses at highly ranked schools in the United States, we uncovered some major shortcomings. Performance was rarely discussed in the courses and was either inadequately defined or left undefined. Although performance was sometimes mentioned as a requirement, no guidelines were provided for specifying this requirement. Students were told either implicitly and in many cases explicitly to treat performance as a late life cycle activity. In most cases performance was viewed as a low-level system or algorithmic problem. Few courses discussed how to measure software performance within an application using instrumentation and profiling. None discussed how to measure resource utilization. SPE was mentioned in only two courses, and one of these incorrectly described the research area. These shortcomings make the case for including software performance engineering in computer science education.

In the second half of the paper, we present an outline of an *undergraduate* SPE course. Unlike most courses which teach performance, ours was not about algorithms or system modelling. The goal of the course was to provide students with a set of skills that they could use to design, build, and maintain software with good performance. Course topics included case studies for motivation, software modelling, measurement and instrumentation, some algorithms, database and web performance, testing and workload generation, seminal research, principles, patterns and antipatterns, and interaction with industry. Projects, homework, and in-class exercises solidified concepts from lecture. We describe one project in detail which involved performance problems with a college homepage.

2. SURVEY

The goal of our survey was to answer this question: What kind of education are undergraduates receiving with respect to designing, building, and maintaining software with good performance?

2.1 Methodology

We focused on undergraduates because a majority of computer scientists do not have a graduate degree in computer science. Data from the National Center for Educational Statistics data regarding postsecondary degree completion for computer and information systems at over 9000 U.S. schools over the past decade supports this claim. Only 20.91% of computer science and information degrees awarded

Year	Associates	Bachelors	Masters	Doctorate
1999-2000	20450	36195	14264	777
1998-1999		Not Available		
1997-1998	13870	26852	11246	858
1996-1997	10990	24768	10098	857
1995-1996	9658	24098	10151	867
1994-1995	10345	24719	10347	894
1993-1994	13158	24499	7730	813
1992-1993	13069	24518	7413	808
1991-1992	12992	24872	6884	775
1990-1991	7677	25083	9324	676
1989-1990	11517	27700	6969	623
Total	123726	263304	94426	7948
%	25.28%	53.80%	19.29%	1.62%

Figure 1: Degree Completion Table

were graduate degrees (see Figure 1) [19].

Because of the vast number of postsecondary schools in the United States, we restricted our survey sample to software engineering and performance courses at highly ranked computer science schools in the United States. We define "highly ranked" as the top 24 research and doctorate schools according to data from Computer Research Association's Taulbee Survey [7]. We believe these schools provide an excellent education for undergraduate computer scientists. According to the 2000-2001 Taulbee Survey, these schools produced 28% of computer science Bachelor degrees at research/doctorate schools and 10% of computer and information systems related Bachelor degrees at all postsecondary schools nationwide.

We visited the online course catalog for each of the top ranked schools, creating a list of courses to investigate based on description and keywords like "performance", "efficiency", and "scalability". For each course, we then reviewed the syllabus and lecture notes of the most recent offering.

The term "performance" had many meanings depending on the goal of the course. For a course on data structures and algorithms, performance meant abstract computational complexity. Although critical to a computer science education, we ignored such courses because they generally presented algorithms in isolation from the rest of the software development process.

Operating systems, computer architecture, or network courses usually viewed performance from a system perspective. In such courses, the focus of performance was on the behavior of hardware and low-level software components in the system. Like algorithms courses, these system elements were generally isolated from applications that use them and from the software development process. Consequently, we eliminated these systems courses from the survey.

Occasionally we encountered a performance course. This type of course always focused on system models of hardware and low-level software components of a computer system, rather than models of a software application. Course content usually covered topics such as probability, statistics, queuing theory, and Markov chains. We felt these performance courses were inappropriate to the survey for several reasons. First, the courses were almost always taught at the graduate level. Second, a lower level systems approach was taken to modelling rather than a software application approach. Third, modelling was the focus of the course, rather than engineering an application with good performance. Finally (and somewhat controversially), after more than a decade of industrial performance engineering experience, we have rarely needed to use advanced system modelling. After discussing modelling with SPE veterans and reviewing the SPE literature, it appears that our modelling experience is not an isolated one.

The majority of courses included in our survey had a software engineering focus. Generally, a university offered a single course for upperclassmen coupled with a large group programming project. Occasionally, however, a university did not offer a dedicated course. Software engineering concepts were introduced to freshman or sophomores in the introductory programming courses instead. These introductory courses were also included in our survey.

We also considered software engineering courses for graduate students. We reasoned that in some circumstances upperclassmen might be able to take these courses. In general, most schools offered a single graduate course, but two universities with software engineering research centers (Carnegie-Mellon's Software Engineering Institute and University of Southern California's Center for Software Engineering) provided a rich selection.

We examined the material of 64 courses that appeared relevant to software engineering and performance. Eliminating algorithm, system, and system modelling courses further reduced our sample to 45. We were unable to obtain on-line notes for several classes leaving a final sample size of 37.

If we were unable to evaluate a survey category for a course, then we left the category entry blank.

2.2 Limitations

Our study had a number of limitations. To classify a course we looked predominantly at the instructor's online lecture notes. We excluded other important material such as projects, exams, quizzes, reading assignments, and textbooks.

Many courses promoted a software engineering methodology, such as Extreme Programming [3] or Agile Software Development [9]. We used the methodology's performance guidelines (if any) to help classify the course.

Although lecture notes provided a framework and summary of lecture, there was a great deal we probably missed by not attending the lectures of the courses we surveyed.

We used document text search tools to look for a small set of performance keywords. We could have looked for more keywords like "size", "capacity", "scalability", "time", "throughput" etc. Many search tools ignore words in images embedded in a document. We may have overlooked some performance related material because of these limitations.

We tried to make our classification system objective, nonetheless, subjectivity crept into the system. For example, we tried to get a "feel" for how performance was presented in relation to other software requirements. We also tried to determine subjectively how significant a role performance played in any case studies.

It could be argued statistically that our sample does not represent the population of all postsecondary schools in the United States. An alternative sample would have been a random selection of software engineering courses from all postsecondary schools in the United States.

2.3 Analysis

An analysis of our survey shows serious shortcomings with respect to the treatment of performance in software engineering courses:

- The majority (64.87%) of software engineering courses spend little or no time on the subject of performance (Section 2.4.1).
- The majority of courses (84.84%) had an inadequate or missing performance definition (Section 2.4.2).
- The majority of courses (82.14%) provided either qualitative or no guidelines for performance requirements (Section 2.4.3).
- The majority of courses (70.59%) treated performance as a subclass of non-functional requirements (Section 2.4.4).
- The majority of courses (72%) advocated a reactive methodology either implicitly or explicitly (Section 2.4.5).
- The majority of courses (74.07%) implied that either low-level system components or algorithmic efficiency had the most influence on application performance (Section 2.4.6).
- Few courses (10.6%) dedicated significant lecture time to the study of real world software engineering projects. However when a case was studied, performance played an important role seventy-five percent of the time (Section 2.4.7).
- Very few courses (16.22%) mentioned any technique for measuring application performance. Manual instrumentation was discussed in these few courses, usually in conjunction with automated profiling. No course included a discussion of the tradeoffs between manual and automated instrumentation. Moreover, no discussion of techniques for measuring throughput and resource utilization was evident in notes for any course (Section 2.4.8).
- Very few courses (16.22%) provided guidance as to where to focus effort to achieve good application performance. Although the performance centering principle was not mentioned in these few courses, Amdahl's Law was often referenced (Section 2.4.9).
- Few courses (27.02%) we examined advocated some form of performance modelling. Of these courses, half used models for performance validation before implementation while the other half used models to diagnose problems in an implemented system (Section 2.4.10).
- Only two out of the 37 courses surveyed mentioned SPE. One of these courses mislabelled the research area as "Software Performance Evaluation" and implied that SPE was a specialized discipline for database-centered and real-time applications [12] rather than a broadly applicable discipline (Section 2.4.11).

2.4 Results

The results of our survey, organized by school and course appear in Figure 6. Several categories are missing from this table due to space constraints and a low number of responses. A summary of these results, with an explanation of each category appears in this section.

1. Lecture Frequency (How frequently was performance discussed during lectures? N=37)

- Frequent (35.14%) - there was at least one entire lecture (or equivalent) dedicated to the topic of performance.
- Intermittent (37.84%) - performance was mentioned occasionally, with the sum total discussing performance equaling several pages of lecture notes.
- Infrequent (27.03%) - performance was rarely mentioned, if at all in the lecture notes.

2. Definition (Did the course provide an adequate definition of performance? N=33)

- Adequate (15.15%) - the definition included response time, throughput, and resource utilization in response to a workload.
- Inadequate (45.45%) - the definition missed one or more of the components (e.g. mentioning response time, but not resource utilization), used confusing vocabulary (e.g. bandwidth instead of throughput), or used redundant and incomplete definitions (e.g. performance: timing, latency, etc.).
- Undefined (39.39%) - the term performance was used in lectures, but left undefined.

3. Requirement Specification (Did the course provide guidelines for defining performance requirements? N=28)

- Quantitative (17.86%) - the lectures provided numerical values for performance requirements such as response time, throughput, and resource utilization.
- Qualitative (71.43%) - the lectures used descriptive terms like "good", "efficient", and "fast" were when discussing performance requirements.
- No Guidelines (10.71%) - the lectures provided no guidelines for defining performance requirements.

4. Requirement Weight (How did performance rank against functional and other non-functional requirements such as reliability, security, usability, etc.? N=17)

- Equal with Functional (29.41%) - performance had the same weight as functional requirements.
- Sub-class of non-functional (70.59%) - performance had minor role in a large hierarchy of non-functional requirements.

5. Methodology (Was the performance methodology proactive or reactive? N=25)

- Proactive (28%) - the course advocated awareness and planning for performance throughout all phases of the development cycle.
- Implicitly Reactive (28%) - although a fix-it-later attitude was not stated explicitly in the course, such an attitude was implied by leaving the discussion of performance to the end of the course, discussing performance during late life cycle activities, discussing performance only in the context of optimization and tuning, or relegating performance to optimizing compilers and hardware.

- Explicitly Reactive (44%) - students were told explicitly to get the system working first, and performance performance was relegated to the end of the software development cycle.
- Knuth Quote (16%) - the course justified an explicitly reactive approach by quoting the following statement from Knuth's classic paper on structured programming:

> *We should forget about small efficiencies, about 97% of the time. Premature optimization is the root of all evil [15]*

Courses in this category were also counted in the Explicitly Reactive category.

6. Influence (What factors influenced the performance of a software application? N=27)

- Holistic (25.93%) - the course took into account many factors including requirements, user perception, commercial off-the-shelf software (COTS), and workload, as well as system and algorithmic issues.
- System (59.26%) - the course viewed performance as a low-level system issue, relegating responsibility to the correct choice of hardware, programming language, and optimizing compiler. Frequently this type of course would include a special section on language specific programming performance "tricks" (e.g. use `StringBuffer` instead of `String` for dynamic strings in Java).
- Algorithm (14.81%) - the course viewed efficient algorithms as having the biggest impact on performance.

7. Case Studies What role did performance play in real world case studies? N=8)

- Major (75%) - performance was a major concern in the case study.
- Minor (25%) - performance may have been mentioned, but was not a large concern in the case study.

For this category, we only considered case studies where the majority of a lecture was dedicated to a case study.

8. Measurement (How was the measurement of performance approached in the course? N=37)

- Manual (5.41%) - the course advocated building instrumentation into the code during development.
- Profiler (3.23%) - the course advocated using an automated profiler to periodically measure performance.
- Both (8.11%) - the course advocated both manual and profiler techniques.
- None (83.78%) - the course provided no guidance with respect to performance measurement.

9. Processing vs. Frequency (Were students given any guidance as to where to focus for application performance? N=37)

The processing vs. frequency centering principle states the product of processing and frequency should be minimized [25]. Amdahl's Law rewords this principle by stating that a performance improvement is limited to the fraction of the time the improvement can be used [1]. Critical use cases drive these principles by identifying which software components are going to be exercised frequently by users and ensuring that these components meet performance requirements.

- Aware (16.22%) - the course reviewed the processing vs. frequency principle or Amdahl's Law.
- Unaware (83.78%) - the course did not review the processing vs. frequency principle or Amdahl's Law.

10. Modelling (Was performance modelling included in the course? N=37)

- Proactive (13.51%) - the course advocated performance modelling portions of the software system before implementation.
- Reactive (13.51%) - the course advocated performance modelling portions of the software system that exhibited performance problems after implementation.
- None (72.97%) - the course did not discuss performance modelling.

11. SPE (Were contributions from the Software Performance Engineering community mentioned during the course? N=37)

- Aware (5.41%) - the course presented information about SPE.
- Unaware (94.59%) - the course did not present information about SPE.

2.5 Conclusions

We have shown that students are not learning the meaning of performance. Therefore, students will have difficulty understanding how to incorporate performance into the software engineering process.

We have shown that the specification portion of software engineering courses concentrate on functional requirements. Functional requirements tend to be qualitative in nature. We speculate that this qualitative focus, coupled with a lack of guidelines with respect to performance requirements, may cause students to conclude that performance requirements should also be qualitative.

We have shown that students are not being taught that performance is important, are not being taught what performance is, and are being given no guidance as to how performance requirements are specified. We believe that the probability that students will concern themselves with performance early in the development cycle is low.

We have shown that many non-functional requirements compete for student attention in a software engineering courses. Figure 2 from Sommerville's popular software engineering textbook illustrates this phenomena [26]. Performance may not be getting sufficient attention in software engineering courses.

We have shown that students are being taught to consider software performance a costly late life cycle activity (see Figure 3 [20]).

We have shown that students may be misled into believing that, for example, rewriting Java math routines in C++ or waiting the 18 months of Moore's Law for an improvement in processing speed will make performance problems disappear. System issues and algorithm choices do impact performance, but they should be part of a holistic view in the context of the software life cycle.

We have shown that students are not being taught basic performance measurements skills. The performance requirements of a software application will be difficult to verify if students do not know how to measure response time, throughput, and resource utilization.

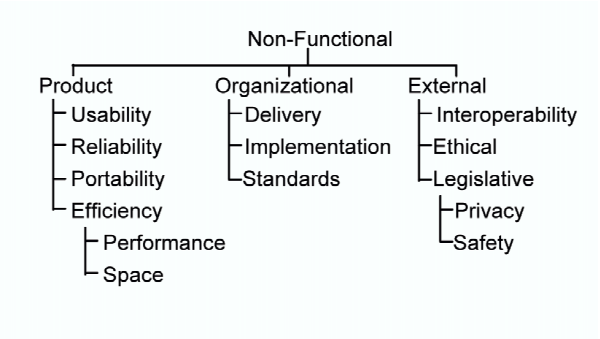

Figure 2: Non-Functional Requirements

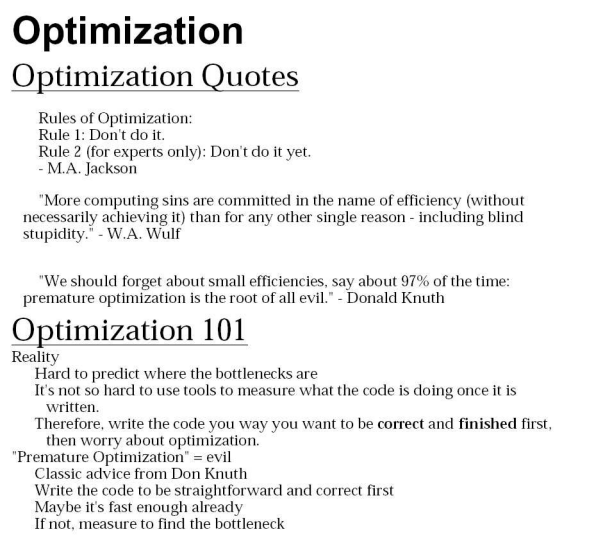

Figure 3: Recent Software Design Course Lecture

We have shown that students are not being taught how to create even basic performance models. At a minimum, software engineering students should learn how to create simple, low cost models to validate performance critical areas of an application before implementation.

Finally, we have shown that there may be a disconnection between the Software Engineering and Software Performance Engineering communities. These courses did not indicate a general awareness of the SPE community, research, and practices in software engineering courses.

3. SPE FOR UNDERGRADUATES

> *[The Software Performance Engineering course] has been applicable to my internship/job. Some of the things we've discussed I've had to do at work. In my experience, no SPE has been performed at the company. I feel better prepared to approach my boss to say specifically what isn't working. I've liked the hands on approach in class working on the models [2].*

One source of software performance problems may be the educational system that produces the software industry's workforce. Our survey results, analysis, and conclusion has support this hypothesis. We need to improve the teaching of performance in software engineering courses. Incorporating software performance engineering as part of the computer science curriculum could satisfy this need.

An SPE education raises some interesting questions:

- Is SPE being taught anywhere?
- Can SPE be a graduate or undergraduate topic?
- What should an SPE course look like?
- What should an SPE course project look like?

We would not presume to provide definitive answers to these questions, but we can offer some opinions which may stimulate discussion.

In the remainder of the paper, the terms "we", "us", and "our" refer to the author, and the terms "students" and "the class" refer to the students in the SPE course.

3.1 Is SPE being taught anywhere?

We answered this question by contacting members of the SPE community via e-mail, and searching the world wide web for courses using the keywords "software performance engineering", "SPE", and a combination of the words like "smith/williams/performance". The results are presented in Figure 4.

SPE courses are offered worldwide to graduates, professionals, and undergraduates.

3.2 Can SPE be an undergraduate topic?

Currently, the majority of SPE courses are offered as graduate or professional education courses.

However, our software engineering survey revealed serious shortcomings in the software and performance education at *all* levels. We also showed that only 20.91% of computer science and information degrees awarded in the past decade were graduate degrees.

For these reasons, we believe that SPE should also be taught to undergraduates.

3.3 What should an SPE course look like?

During the Spring 2003 semester at Stonehill College, we taught an SPE course for *undergraduates*. The goals of our course were to teach students how to design, build, and maintain software with good performance.

Stonehill is a small (2500 students), comprehensive, undergraduate, U.S. college located 20 miles south of Boston. The computer science department has 50 majors and three faculty. Computer science is considered one of the Stonehill's most challenging majors. Most of these majors plan on a career in software development, quality assurance, or software management.

We wanted a textbook that would help students develop skills for building performance into an application at every stage of the software life cycle. We chose Smith and William's relatively recent *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* as the textbook for the course [23]. In addition to being written by pioneers in the field, the book was compatible with our course goals as stated above. Although system modelling plays an important role in SPE, we did not want a text that focused on probability, statistics, queuing theory, and Markov chains.

For the most part, we were very pleased with the textbook, although it was not always a perfect fit. Because

School	Course Name	Course Number	Semester	Level	Website
L & S Computer Technology	Solving Performance Problems Quickly and Effectively		S2003	P	http://www.perfeng.com/psclass.htm
University of Helsinki	Software Performance Engineering	58153003-1	S2003	G	http://www.cs.helsinki.fi/u/verkamo/spe/spek2003_eng.html
Carleton University	High-Performance Software	94511	W2003	G	http://www.sce.carleton.ca/courses/94511/
University of Colorado	Software Engineering of Multi-Program Systems	ECEN 5043	F2002	G	http://ece-www.colorado.edu/~swengctf/swengmulti/info.html
University of Durham	Performance Engineering	Conc&PE-PE	2002/03	UG	http://www.dur.ac.uk/nigel.thomas/cpe/
Performance Dynamics Company	Guerilla Planning		S2003	P	http://www.perfdynamics.com/
American University at Armenia	Enterprise Computing Applications	CIS 270	F2001	G	http://www.aua.am/aua/masters/ce/courses/270.pdf
West Virginia University	Software Performance Engineering	CS793	S2003	G	http://www.csee.wvu.edu/~katerina/Teaching/CS-793-Fall-2002/
George Mason University	Computer System Performance Evaluation	CS 673	S2003	G	http://cs.gmu.edu/~menasce/cs672/cs672.html
University of Connecticut	Software Performance Engineering	CSE 321	F2002	G	http://www.engr.uconn.edu/~reda/
University of Arkansas	Web Capacity Planning		Su1999	G	http://www.csce.uark.edu/~aapon/courses/capacityplanning/
Michigan State University	Computer System Performance Evaluation	CSE807	S2003	G	http://www.cse.msu.edu/~cse807/
Stonehill College	Software Performance Engineering	CS 399	S2003	UG	http://www.cs.stonehill.edu/compsci/SPE
National Technological University	Software Performance Engineering	SE 767-NT	S2003	G	http://www.ntu.edu/ac/accourse.asp?term_id=2003-3&CID=SE+767%2DNT

Figure 4: Software Performance Engineering Courses. LEVEL refers to the level at which the course is taught. UG = undergraduate, G = graduate, P = professional.

the target audience was a professional one with broad technology exposure, the class sometimes found the text a bit inaccessible. The book also lacked exercises and had a few technical errors.

Like courses in many U.S. schools, courses at Stonehill College run three hours per week over a fifteen week semester. We organized the course as follows:

Introduction/Motivation (1 Week)

The first two chapters of *Performance Solutions* formed the backbone of this unit. These were supplemented with case studies from the text, our own experiences, and articles uncovered by students.

During the semester one student contributed an amusing article about a real-time seafood auction website that failed due to performance problems:

> *It's important for an auction site to operate smoothly if unsold products will become worthless hours after the site fails. It is especially vital if those idle products will quickly begin to smell [4].*

Cases studies played an important role throughout the course, prompting one student to comment:

> *I especially like the real world scenarios you have shared with the class. For example, the story you shared about the Staples reporting system that took days to run and was pared down to less than one hour [2].*

Performance Modelling (4 Weeks)

Modelling is critical to any effort involving performance, but we limited the unit to four weeks to prevent modelling from dominating the course.

We wrestled with the question of what kind of modelling should be taught to undergraduates. To help us with the question we considered a survey on the state of the practice of performance engineering, from Carnegie Mellon's Software Engineering Institute which reported:

> *Adoption of performance engineering and its integration into software engineering will be facilitated by the generation of a simplified conceptual model for performance. This model should foster communication between the performance specialist and others, and make performance engineering more accessible to the nonspecialist [14].*

We also considered the recommendation from the SPE community that simple models can be very effective at identifying performance problems, especially early in the software life cycle [23].

We also considered our experience of over a decade of industry successes using simple non-contention models of application performance.

Based on these considerations, we had some guidance for the kind of modelling that should be taught to undergraduates. We adopted the simplified modelling approach taken in *Performance Solutions*. UML use cases and sequence diagrams were used to describe software applications. Sequence diagrams were converted into software execution models further translated to system execution models. We covered simple queueing models at the end of the unit to give the students an appreciation for contention.

Modelling concepts were reinforced with in-class exercises using internet chat and e-mail which the students used on a daily basis. For one of the large course projects, students modelled a sophisticated on-line auction system that they had built in the previous semester's database course. The students were told that the system should support 10 million registered users, 1 million items, and 100,000 live users over a 1 hour period with an average response time of 2 seconds or less for a user action. The project also involved a capacity planning exercise in which one student uncovered a critical design flaw in that added tens of thousands of dollars to the final system cost.

Measurement (1 Week)

In our experience few computer scientists have demonstrated the ability to measure software application performance. The course textbook provided an excellent overview of measurement tools and techniques. We reinforced this overview with in-class exercises and extensive projects outside of class. Tools used included using Microsoft's `perfmon` utility, Microsoft's VC++ automated profiler, VC++ applications, and the ORACLE relational database.

Algorithms (2 Weeks)

One of the most gratifying aspects of SPE for us has been a synergy with the analysis of algorithms. We used this unit to revisit some of the classic sorting, searching, and graph algorithms from an SPE perspective. The abstract complexity of a bubble sort and quicksort can be made concrete by measuring response time and resource utilization for different workloads.

Students learned to recognize linear, logarithmic, polynomial and exponential complexity by measuring and analyzing executing applications. Students also learned about

the Superficial Algorithmitis antipattern [18]. Sometimes an algorithm doesn't behave as theoretically expected in an actual system. For example, the worst case running time of a quicksort may actually be more accurately described as the average case running time because partially sorted data are fairly common.

Database Performance (1 Week)

Previously students had been taught the searching and query processing algorithms used by a modern relational database. In this SPE unit, students were taught how to use commercial query tuning and database monitoring tools to anticipate, improve and maintain database performance. We chose ORACLE because of its market penetration. Students were also introduced to the Circuitous Treasure Hunt [24] and Sysyphus [11] performance antipatterns.

Web Programming (2 Weeks)

Web applications have a large potential user base and interesting performance challenges . We used this unit to introduce students to three-tiered programming (browser, web server, and database server). The two weeks were intended to give the students the skills that they would need to complete the final project.

Testing and Workload Generation (1 Week)

This unit was an introduction to automated testing and load generation using commercial tools like Mercury Interactive's LoadRunner and Segue's SilkPerformer. Students learned how to use the tool to create workloads based on critical use cases.

Research (1 Week)

We wanted to include a discussion of research important to both software engineering and performance. Topics were restricted to material accessible to undergraduates. Our list included Smith's classic paper on SPE [23], Brook's classic on software engineering [5], CMU's CMM for software engineering [8] and the SPE community response [21].

Industry Visits/Talks (1 Week)

Contact with industry made topics in the course more concrete.

The class attended a software engineering seminar given by several programmers at General Dynamics. At the end of the talk, we asked how the programmers approached performance during the software life cycle and if there had been any recent performance challenges:

> *We were told our by senior architects to ignore performance during development and just get the system to work. In the end, however, performance was the biggest problem that we had on the last project [13].*

The seminar provided interesting class discussion.

Principles, Patterns and Antipatterns (1 Week)

We concluded the course with a review of basic SPE principles and patterns from the textbook. Students also read Smith's paper on performance antipatterns [24] and we included some of our own antipatterns [11, 18]. The review provided performance guidelines students use in industry.

3.4 What should an SPE course project look like?

> *I hear and I forget, I see and I remember, I do and I understand* - A Chinese Proverb

Course projects and in-class exercises reinforced lectures and the textbook:

- Stonehill College Homepage - students discovered a performance problem with the college homepage, constructed a performance model to understand the problem, verified the model experimentally, and determined a low cost solution that was implemented by the college. This project is covered in more detail in the next section.
- Online Auction System Performance Model - students modelled a sophisticated on-line auction system that they had built in the previous semester's database course.
- Instrumentation API - students wrote a high resolution instrumentation class in C++ and compared measurements against an automated profiler for several small applications.
- Monkey Typing - students used their modelling skills and instrumentation API to diagnose and correct a series of performance problems in a more sophisticated application. The program simulated a monkey typing randomly on a keyboard for a thousand years, and extracted words to form prose.
- Student Registration Website - students created a three-tiered application (browser, web server, database backend) for online course registration. For a given set of hardware, students demonstrated the ability of the system to scale to 100, 1,000, and 10,000 users.

4. SAMPLE SPE COURSE PROJECT: WEBSITE HOMEPAGE

> *The website project was extremely interesting and almost enjoyable :) I liked to be able to take something I look at every day and figure out how it works and how to fix its problems [2].*

4.1 Background

During the 2002-2003 school year, Stonehill contracted with a software firm to rework the college's website with the goal of improving content, usability and consistency.

Since performance can also play an important role in a user's web experience, we were curious about impressions of performance with the existing system, and the college's future website plans. In hallway conversations with students, faculty, and staff we learned that there was a performance problem with the current site.

Users complained that the homepage took too long to display. When pressed for a quantitative measurement, users gave us widely varying response times from 10 seconds to more than a minute. These widely varying response times were consistent with prior experience on projects involving poorly performing interactive systems. We knew that when the responsiveness of a system perceived to be interactive exceeds a threshold (5-10 seconds), it becomes difficult for the user to quantify the response time other than to state that it is slow.

The problem was exacerbated because hundreds of computers issued by the school had the web browser configured with the college homepage as "home". Each time the

browser was started, the college homepage was automatically displayed. The majority of the user population was not technically sophisticated enough to change the default.

The project manager for the website redesign was aware that there were some performance problems, but was hopeful that they would disappear when the new site was deployed. A belief that a new system can remove performance problems simply by being new was another phenomenon that we had seen in prior projects.

4.2 The Scientific Method

Because most of the class had also experienced problems with the college homepage, the stage was set for the project.

The class began the project by speculating on the cause of the problem. Since some students were unfamiliar with the architecture of a web system, the class spent some time discussing how a web server, network, and browser work to display a web page. One student suggested that a java applet used by the page for navigation could be the culprit. Another suggested that the large number of browsers pointing to the webpage put a drain on the server or the network. Someone countered these arguments by pointing out that the web page displayed very quickly on computers in the classroom and that maybe there wasn't a problem at all.

In our experience, speculation or "developer intuition" is common when there is a performance problem with a system. With little or no evidence developers use this intuition to "correct" performance problems, sometimes to the detriment of the system.

A better way to approach performance problem is to use the scientific method:

- Observe and describe the performance problem. The description should also evaluate the consequences of the problem.
- Formulate a hypothesis to explain the performance problem. The hypothesis is usually some form of a performance model. The model quantitatively predicts the results of new observations.
- Verify the hypothesis with experiments.

4.3 Observation and Description of Performance Problem

With this scientific framework for investigation, the class focused on observation and description of the Stonehill College homepage performance problem.

What were the consequences of the homepage performance problem? Obviously students, alumni, faculty, administration and staff used the page to get information about campus news and events. This captive audience would probably tolerate some page delay if the information was important (e.g. class cancellation due to weather). Besides, the current webpage had been in use for several years with only minor complaints.

A prospective student's first impression of the college also came from the homepage. "The latest market research shows that 70% of all college-bound high-school seniors began their college search on the web, and those virtual tours came second only to actual campus visits in luring students [10]."

In her project report, one student described the serious consequences of the performance problem in a nightmare scenario for any college admissions office:

> *Picture this, an 18-year-old male is sitting in front of his computer in Cleveland, Ohio searching the Internet for the college that is just right for him... When he clicks on the link to Stonehill's website, it takes between twenty-five and forty-five seconds to fully load. After waiting for approximately twenty seconds, the young male stopped the site from loading and went back to the list to choose another college to learn about [16].*

The homepage displayed quickly on computers in the lab, but some had experienced a sluggishness on other computers. What was different? Computers in the student dormitories and at home seemed to take longer rendering the homepage.

Although dormitory computers had high speed access to the internet (6 Mbps shared bandwidth), students were constantly complaining about performance problems with web and chat programs. Over a period of years, the college information technology department had taken measures to isolate the dorm network from the rest of the campus to control the high bandwidth demands for music and video files made by student computers. Using a port filter, file sharing programs were blocked from 7AM to 9PM weekdays. With filtering, network bandwidth utilization averaged less than 30%. Without filtering, utilization jumped to almost 100%. The homepage displayed quickly when the port filter was active, and slowly when inactive.

What about access to the homepage from off-campus? Some homes also have high speed access to the internet via cable modems, satellite, or dsl. Others have slower dial-up access via a telephone modem. The homepage displayed quickly using a home computer attached to a cable modem. The homepage displayed slowly using a telephone modem.

Quick and slow were qualitative terms. The class also needed to establish *quantitatively* an acceptable response time for the page.

What about throughput and resource utilization? On the server side, these two represented important performance measurements. After a conversation with the college network administrator, the class was given some assurances: that the website received at most five thousand hits per day, that there was ample bandwidth on the academic side of the network to handle this throughput (3Mbps outbound), and that the CPU, disk, and memory utilization on the web server was very low. On the client side, it was difficult to measure CPU, disk, memory, and network utilization during a page load in the classroom because pages loaded quickly.

Based on the observations that the class made about when and where the homepage displayed slowly, the class suspected the problem was related to the size of the web page and the client-side network bandwidth. The page rendered quickly in environments where there was ample bandwidth (e.g. cable modem), and slowly when bandwidth was constrained (e.g. telephone modem).

4.4 Hypothesis and Experiments

The class measured the size of the components that made up the web page (178KB on average). Despite browser-side caching, there was still some overhead (43KB on average) associated with page refresh because of a script which selected a random image from a library of campus photographs to give the page a fresh look.

Based on these measurements, the class developed a simple performance model that predicted the expected hompage load times for common network bandwidths. The model predicted an initial page load time of 31.73 seconds and refresh page load time of 7.63 seconds when using a 56Kb modem.

The class then conducted an experiment which measured actual load times for the web page using a 56Kb modem. The measurements agreed with the performance model that had been constructed. The average initial page load time was 31.35 seconds with a refresh of 8.44 seconds!

It appeared that the class was headed in the right direction, but more questions arose. First, what was the network bandwidth for a typical web page visitor? According to Neilsen/Netratings in December 2002 narrowband (14.4Kb - 56Kb) users comprised the bulk of the U.S. online population with 74.4 (69%) million users, while high speed internet access accounted for 33.6 (31%) million users [6].

If the majority of visitors had narrowband access, then was the initial load and refresh of the web page too slow? "Slow" was a qualitative term, and the class discussed how we might make this more quantitative. Some research has been done in this area, but it was inconclusive [22]. In discussions with SPE colleagues, eight seconds had been mentioned as the threshold for web page responsiveness and number was suggested to the class. Students felt that eight seconds was too high.

Searching the internet students found a of number web design sites with performance advice. Many sites advised designers to keep the web page size below some threshold, although there was no agreement on what this threshold should be. One site suggested a response time threshold based on holding one's breath!

The class wanted something more quantitative and less anecdotal so an experiment was designed. Subjects were asked to visit a web page replicating the Stonehill homepage. The web server introduced a random delay in the web page before sending it to the browser (1,2,5,10,15, and 30 seconds). Without knowing how long the web page took to display, subjects were asked to rate the responsiveness of the page on a scale from 1 to 5 (very fast, fast, acceptable but could be faster, slow, very slow). The test was repeated ten times for each subject. Although the subjects were always connected to the web server via a local high speed network, they were told that for the first five viewings a high speed internet connection was being used, and for the second half a 56Kb modem was used.

Four hundred and fifty observations were recorded with an average of 37.5 measurements for each browser delay category. The average responsiveness rating for each delay appears in Figure 5. The standard deviation was consistently between .7 and 1.0 for each category.

4.5 Results and Analysis

The results from the browser delay experiment provided some interesting information for the class. First, user expectation regarding performance was the same whether using a broadband or narrowband connection. Second, the threshold appeared to be somewhere between 5 and 10 seconds. Applying a rough curve fit to the in Figure 5 narrowed the threshold to between 7 and 7.5 seconds which was inline with SPE common wisdom.

It was now clear that Stonehill's web page exceeded the threshold for acceptable performance both for initial load and refresh. The problem with the initial load was that there were simply too many incidental scripts and images that had to be downloaded into the browser for a first time visitor. The performance of the initial load was consistently rated "very slow" by subjects. This slowness posed a significant risk that the page might be abandoned by a first time visitor/prospective student. The class recommended that the total size be reduced from 176KB to 40KB for the new web page.

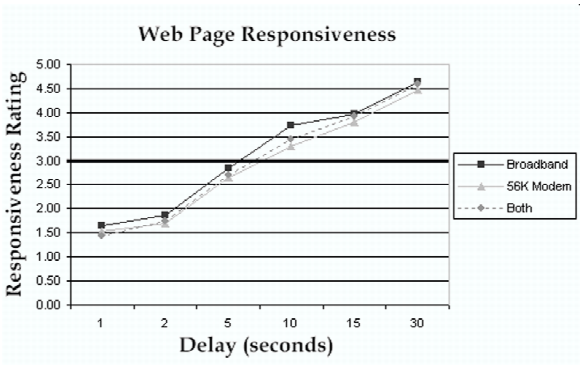

Figure 5: Web Responsiveness Experiment Graph

The problem with the refresh was that a browser side script generated the file name of a random image to download on each refresh. The large library of random images (over twenty averaging 46KB each) effectively disabled the browser cache.

The goal of the random images was to give the website a "fresh" look each time the website was visited. The class brainstormed how to preserve this functional requirement while still meeting the performance threshold. Several ideas were proposed:

- server side: provide new image every week/day/hour
- browser side: offer a "next" link that would allow users to view additional images on the web page
- browser side: computes a filename for a new image every week/day/hour

Students compiled the results of the research, experiments, and analysis into a project report. The best reports were sent to the project manager for the website redesign. After reading the reports, the manager explained that the new homepage would have fewer scripts and images, but would still have an image rotation scheme. He recognized that image rotation might cause a performance problem, but did not want to lose the functionality - a conflict that occurs often in software performance engineering:

> *What is the right balance between as Nick puts it in his paper, "presentation" and "performance" - If you feel it could be in the scope of your class, I would love to have you/your students provide some solid recommendations on the balancing of presentation and performance - I know that your course focus is performance, but the real value of course lies in maximum performance with minimum impact on presentation.*

The class was given early access to the new homepage and conducted another series of observations, modelling, experiments, and analysis. The page was much simpler, but at 95KB the page was still too large resulting in a 17.14 second initial load. Page refresh was almost immediate because all of the images used in rotation to keep the page fresh were downloaded during the initial load. Almost 70% of the page size was traced to a 65KB Macromedia Flash file containing the images that were used in rotation. For the initial deployment this file contained 5 images, but the director told the class there could be as many as 15 in the future. The class's performance model estimated that a 15 image flash file would result in a 223KB web page and an initial load time of 39.89 seconds for a 56Kb modem.

Another report was sent to the project manager based on an analysis of the new homepage. The report recommended abandoning the Flash file approach to image rotation. As an alternative, a detailed explanation of a browser side script which computed a new image filename on a periodic basis using the browser's system time, the number of available images, and a modulus function was provided.

The new homepage was changed based on the class's recommendations, and now has an average size of 45KB with an initial download time of 8.14 seconds and an almost instantaneous refresh time.

5. FUTURE WORK

We plan to make several refinements the next time we teach the SPE course. We were unable to complete several important course units (database performance, web performance, testing and workload generation, research, and the final project) due to time constraints. We hope to cover more material next time by balancing in-class exercises with more outside homework. We also plan to incorporate new discoveries from the rapid advances being made by the SPE community.

We plan to use our software engineering course survey to stimulate discussion between the SE and SPE community and improve the presentation of performance in software engineering courses. We would like to investigate how software engineering textbooks and popular methodologies approach performance in a future survey. We also intend to integrate SPE topics from our course into our senior capstone software engineering course next spring. Integrated topics will include: UML to System Execution modelling, measurement, and testing/workload generation.

We plan to investigate the user perception of responsiveness by expanding upon the SPE class web page responsiveness experiment. We will repeat the experiment across an array of application tasks and user populations. The goal of these new experiments will be a set of quantitative performance guidelines for responsiveness.

6. CONCLUSION

We began this paper by proposing that one cause of the problems industry experiences with software performance may be the educational system producing the programming workforce.

We conducted a survey examining how the topic of performance was presented in software engineering courses at highly ranked computer science schools in the United States. After viewing the online lecture notes for 64 courses, we uncovered major shortcomings. Performance was rarely discussed and was either inadequately defined or left undefined. Although performance was sometimes mentioned as a requirement, no guidelines were provided for specifying this requirement. Students were told either implicitly and in many cases explicitly to treat performance as a late life cycle activity. In most cases performance was viewed as a low-level system or algorithmic problem. Few courses discussed how to measure software performance within an application using instrumentation and profiling. None discussed how to measure resource utilization. SPE was mentioned in only two courses, and one of these incorrectly described the research area.

These shortcomings make a case for including software performance engineering in computer science education. In the second half of the paper, we presented the outline for an SPE course we developed and taught specifically for undergraduates. We chose undergraduates because only 20.91% of computer science and information degrees awarded in the past decade were graduate degrees.

Unlike most courses which teach performance, ours was not about algorithms or system modelling. The goal of the course was to provide students with a set of skills they could use to design, build, and maintain software with good performance. Course topics included case studies for motivation, software modelling, measurement and instrumentation, some algorithms, database and web performance, testing and workload generation, seminal research, principles, patterns and antipatterns, and interaction with industry.

Projects, homework, and in-class exercises solidified concepts from lecture. We described one project in detail which involved performance problems with a college homepage. The basic problem was trivial: the web page was too large for narrowband network connections. However, the work that had to be done to detect, make a case for correction to management, and propose a solution that minimized the impact to functionality and schedules was a great lesson in Software Performance Engineering.

The computer science educational system may be one source of the performance problems that plague the software industry. Teaching Software Peformance Engineering to *undergraduates* is an important application of the principle that early performance intervention in reduces the likelihood of costly performance problems.

7. ACKNOWLEDGEMENTS

This work was funded in part by a Stonehill College Summer 2003 Research Grant. We would like to thank Ralph Bravaco, Virginia Polanski, and Shai Simonson for comments and suggestions. We would also like to thank the students of the Stonehill Spring 2003 CS399: Software Performance Engineering class for their hard work and feedback during the course.

The title of this paper was inspired by *Lies My Teacher Told Me: Everything Your American History Textbook Got Wrong* by James W. Loewen [17].

8. REFERENCES

[1] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, pages 483–485, 1967.

[2] Anonymous. Cs399 software performance engineering. Mid-semester course evaluation Stonehill College, March 2003.

[3] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1st edition, October 1999.

[4] J. Boyd. Seafood auction fights back: User confidence returns slowly after glitch. *Internet Week*, page 19, July 2001.

[5] F. P. Brooks. *No silver bullet—Essence and accident in software engineering*. 1986. Reprinted as Chapter 16 of Brooks, F. P. 1995. The Mythical Man-Month, Anniversary Edition, Addison-Wesley, Reading, Mass.

[6] M. Bumatay and M. Heineman. Broadband usage climbs 59% while narroband usage declines. *Neilsen/Netratings News Release January 15, 2003*. `http://www.nielsen-netratings.com`.

[7] R. E. Byrant and M. Y. Vardi. Taulbee survey. *Computer Research News*, 14(2):4–11, March 2002.

[8] M. Chrissis, B. Curtis, and M. Paulk. Capability maturity model for software, version 1.1. Technical Report 24, Software Engineering Institute, Carnegie-Mellon University, February 1993 1993.

[9] A. Cockburn. *Agile Software Development*. Addison-Wesley, 1st edition, December 2001.

[10] R. Dezember. Campus tour is just a click away — more students winnow lists of possible colleges after traveling the internet. *Wall Street Journal*, page D2, October 2002.

[11] R. F. Dugan, E. P. Glinert, and A. Shokoufandeh. The sysyphus database retrieval software performance antipattern. In *Proceedings of Third International Workshop on Software and Performance (WOSP2002)*, Rome, Italy, July 2002. ACM Press.

[12] D. Garlan. *17-655-A: Architectures for Software Systems*. Carnegie-Mellon University, Spring 2002. `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/17655-s02/www/`.

[13] A. Johnson and M. Menard. General dynamics: Software engineering in the workplace. Stonehill college department of computer science seminar, General Dynamics, Easton, MA, March 2003.

[14] M. H. Klein. State of the practice report: Problems in the practice of performance engineering. Technical Report CMU/SEI-95-TR-020, Software Engineering Institute, Carnegie Mellon University, February 1996.

[15] D. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, December 1974.

[16] K. Leonard. Spe project one. Project report, CS399 Software Performance Engineering, Department of Computer Science, Stonehill College, Spring 2003.

[17] J. Loewen. *Lies My Teacher Told Me: Everything Your American History Textbook Got Wrong*. Touchstone Books, Berkeley, California, 1996.

[18] J. T. McKenna and R. F. Dugan. The superficial algorithimitis, blind spot, and memory vs. cpu software performance antipatterns. In *Poster Session: Eighth Annual Consortium for Computing Sciences in Colleges Northeastern Conference, Rhode Island College, April 25-26, 2003*, Rhode Island College, Providence, Rhode Island, April 2003.

[19] F. Morgan. *Integrated Postsecondary Education Completions Data Files (IPEDS)*. National Center for Educational Statistics, 2003. `http://nces.ed.gov/Ipeds/completions.asp`.

[20] N. Parlante. *CS108 Object Oriented Design: Java Implementation and Performance Lecture*. Stanford University, Winter 2003. `http://www.stanford.edu/class/cs108/handouts`.

[21] A. Schmietendorf, A. Scholz, and C. Rautenstrauch. Evaluating the performance engineering process. In *Proceedings of the second international workshop on Software and performance*, pages 89–95. ACM Press, 2000.

[22] B. Schneiderman. Response time and display rate in human performance with computers. *Computing Surveys*, 16(3):265–285, 1984.

[23] C. U. Smith. Software performance engineering. In *Proceedings of Computer Measurement Group International Conference XIII*, pages 5–14, New Orleans, Louisianna, December 1981. Computer Measurement Group.

[24] C. U. Smith and L. G. Williams. Software performance antipatterns. In M. Woodside, editor, *Proceedings of Second International Workshop on Software and Performance (WOSP2000)*, Ottawa, Canada, 2000. ACM Press.

[25] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, Reading, Massachusetts, 2001.

[26] I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 6th edition, 2001.

	Course			Lecture Frequency			Definition			Requirement Specification			Requirement Weight		Methodology				Influence		
School	Course Name	Course Number	Semester	Frequent	Intermittent	Infrequent	Adequate	Inadequate	Undefined	Quantitative	Qualitative	No Guidelines	Equal w/ Functional	Subclass of Non-functional	Proactive	Implicitly Reactive	Explicitly Reactive	Knuth Quote	Holistic	System	Algorithm
Cornell	Software Engineering	CS501	S2003	1			1					1				1				1	
MIT	Laboratory in Soft. Engineering	6.17	S2003		1				1			1					1				
MIT	Principles of Software Systems	8.826	S2002	1				1		1			1			1				1	1
MIT	Soft. Eng. for Internet Apps	6.171	F2002		1		1			1			1		1					1	
Stanford	OOP Systems Design	CS108	W2002	1					1								1	1		1	
Stanford	Software Project	CS194	S2003			1			1			1									
Stanford	Internet Tech. and Strategies	CS241	S2003	1				1				1				1					
UC Berkeley	Software Engineering	CS169	S2003		1			1						1		1					
CMU	Methods of Soft. Dev.	CS 17-65	S2003			1			1			1									
CMU	Arch. for Soft. Systems	CS 17-755	S2002		1			1				1		1						1	
CMU	Models of Soft. Systems	CS 17-651	F2002			1															
CMU	Intro. to Comp. Systems	CS 15-213	S2003	1				1				1				1				1	
Univ. Washington	Software Engineering	CSE 403	S2003		1			1				1		1		1	1	1			
Univ. Washington	Software Engineering	CS 503	W2002			1			1			1		1							
CalTech	Intro. to Soft. Engineering	CS 3	S2003			1			1												
Univ. Illinois Urb-Chmp	Software Engineering I	CS 327	F2002		1			1				1		1							
Univ. Illinois Urb-Chmp	Software Engineering II	CS 329	Unable to access website																		
Univ. Wisconsin			No courses on software engineering																		
Princeton	Intro. to Prog. Systems	CS217	S2003		1			1				1					1			1	
Princeton	Advanced Progr. Techniques	CS333	S2003		1				1			1					1				1
U.T. Austin	Software Engineering	CS373	S2003	1											1				1		
Brown	Intro. to Soft. Eng.	CS32	S2003			1			1			1		1						1	
Brown	Software System Design	CS190	S2002	1				1			1						1			1	1
Rice	Applied Alg. and Data Str.	COMP 314	S2003	1				1				1					1			1	1
Univ. Michigan	Software Enginnering	EECS 481	Unable to access website																		
Univ. Michigan	Software Engineering Tools	EECS 581	Unable to access website																		
UC Los Angeles	Software Engineering	CS130	W2002		1				1		1		1		1						
Harvard	Computing Hardware	CS141	F2002	1			1	1												1	
NYU	Software Engineering	V22.0474-001	S2003		1				1			1					1	1		1	
UM College Park	Software Engineering	CMSC 435	S2003		1			1				1		1					1		
UM College Park	Internet-Scale Soft. Eng.	CMSC 838P	S2003			1															
Yale	Data Strs.and Progr.Techniques	CPSC 223	S2003		1												1				
Univ. Pennsylvania	Soft. Design/Eng.	CSE 350	S2002			1														1	
Univ. Pennsylvania	Software Engineering	CIS 573	Lectures unavailable online																		
UMASS Amherst	Intro. to Software Engineering	CS320	S2003		1			1				1		1			1				
Columbia	Adv. Software Engineering	COMS W4156	F2002	1				1		1			1		1				1		
Columbia	Software Engineering	CS3156	F2001		1				1			1					1	1		1	
Univ. Southern Calif.	Software Engineering I	CSCI577a	F2002	1			1			1				1	1				1		
Univ. Southern Calif.	Software Engineering II	CSCI577b	S2003			1			1			1		1							
Univ. Southern Calif.	Software Architecture	CS578	S2003	1				1			1			1	1				1		
Univ. Southern Calif.	Des./Constr.of Large Soft. Systems	CS 477	S2003			1			1			1		1		1			1	1	
UC San Diego	O-O Software Design	CSE 111	Lectures unavailable online																		
UC San Diego	Software Engineering	CSE 112	Lectures unavailable online																		
UC San Diego	Princ. of Software Engineering	CSE 210	Lectures unavailable online																		
UC San Diego	Software Testing and Analysis	CSE 211	Lectures unavailable online																		
UC San Diego	Adv. Topics in Soft. Eng.	CSE 218	W2003	1			1			1			1		1				1	1	
SubCategory Total				13	14	10	5	15	13	5	3	20	5	12	7	7	11	4	7	16	4
Percentages				35.14%	37.84%	27.03%	15.15%	45.45%	39.39%	17.86%	10.71%	71.43%	29.41%	70.59%	28.00%	28.00%	44.00%	16.00%	25.93%	59.26%	14.81%
Category Total						37			33			28		17			25				27

Figure 6: Course Survey Table