Informatics: A Novel, Contextualized Approach to Software Engineering Education

André van der Hoek, David G. Kay, and Debra J. Richardson

Department of Informatics, Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3440 USA
{andre,kay,djr}@ics.uci.edu

Abstract. Over the past decade, it has been established that a good education in software engineering requires a specialized program of study different from traditional computer science programs. What should constitute such a specialized program of study, however, is still a matter of debate. Here we bring to this debate a new perspective that describes how we believe software engineering education should be framed, namely through the *context* in which software eventually is placed. That is, we must study software *and* information, development *and* design, technical *and* social issues, synthesis *and* analysis. At UC Irvine, we have designed and now offer a program of study that provides this focus – a four-year B.S. degree in Informatics. In this paper, we present our view of software engineering education, the principles underlying our Informatics curriculum, an overview of the curriculum itself and its pedagogy, some reflections on our experiences to date, and a concluding list of challenges that our approach addresses and that are critical for any approach to software engineering education.

1 Introduction

While software engineering has been a discipline for several decades, software engineering education is only now coming into its own. It is no longer considered adequate for software engineering to be relegated to one or two courses that are part of a traditional computer science education. Rather, it has been established, both philosophically and through experience, that an effective software engineering education requires a specialized program of study [1]. Such specialized programs are now appearing at universities around the world.

But deciding on exactly what to teach and how to teach it is difficult. Such decisions must be guided by an overall philosophy of how to frame the material for students. The McMaster University undergraduate degree in software engineering exemplifies one such framing, which is built from a mathematical and engineering perspective [2]. Based on a first year that rigorously introduces students to mathematical and engineering principles, subsequent years introduce a host of software engineering topics by rooting them in the underlying theory (while still addressing the practical side of the discipline, naturally). This kind of framing is not uncommon, since many view software engineering as an intrinsic engineering discipline that must therefore share its principles with other engineering disciplines.

A second framing is provided by the ACM/IEEE curricular guidelines for software engineering education [3]. These guidelines break down software engineering into a set of closely-related fundamental knowledge areas, each refined into detailed topics to be addressed by a software engineering curriculum. The fundamental knowledge areas focus explicitly on the topic of software construction, covering both technical aspects of software design and implementation and managerial aspects of the broader process involved. Informally examining various software engineering degree programs seems to indicate that, compared to the mathematical and engineering approach described above, the ACM/IEEE guidelines are the more widely adopted model curriculum to date.

In this paper, we present a new, third framing of a software engineering curriculum, one that we have instituted at UC Irvine in a new four-year degree program, a Bachelor of Science in Informatics. What distinguishes our approach is a central focus on *the context in which software will eventually be situated*. That is, we root our approach in the personal, organizational, and societal realities of software. This broadens software engineering education from a focus on software only to a focus on software and on the information that the software manages, from development only to development and design, from technical issues only to technical and social issues, and, finally, from synthesis only to synthesis and analysis. The first factor of each of these four pairs (i.e., software, development, technical issues, and synthesis) tends to be the focus of existing software engineering programs. In our Informatics major, this focus is complemented by an equally strong focus on the other four factors: information, design, social issues, and analysis. The result is a broader kind of software engineering education that we put in the appropriate context of designing solutions, incorporating human and organizational needs, and taking a multi-disciplinary approach to the field.

The remainder of this paper is structured as follows. In Section 2, we further detail our vision of Informatics and its underlying principles as an alternative way to frame software engineering education. Section 3 presents our curriculum and Section 4 the various pedagogical issues involved in delivering the curriculum. Section 5 reflects upon our experiences to date and Section 6 presents some challenges we have encountered that, we believe, must be addressed by any effective SE curriculum. We conclude in Section 7 with our plans for future work and our hopes for the future of software engineering education.

2 Our Perspective and Principles

We begin the discussion of our perspective on software engineering education with a look at the ACM 2005 Computing Curricula document [1]. It attempts to provide perspective by partitioning the broad field of computing into several subdisciplines; we reproduce their result in Figure 1 and refer the reader to the CC 2005 document for their definitions of the subdiscipline boundaries. We note that software engineering has achieved a place, but we also note the lack of connection between software engineering and information technology. This gap may exist because computer science schools tend to offer the majors shown towards the left of the figure while business schools tend to offer those towards the right. One might even contend that this reflects an appropriate separation of concerns. However, we take the opposite view. We argue that the *context* in which the software will be situated profoundly affects the nature of software and of the processes we use to create it. Figure 1 exposes this gap; in Figure 2 we correct it, completing the picture and illuminating the core principle of our approach: explicit consideration of context is essential, both in the practice of software engineering and in the education of its practitioners.

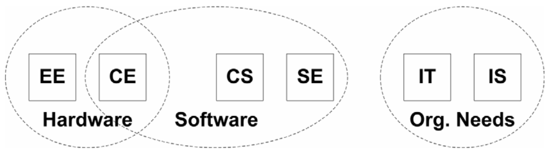


Figure 1. Original Division of Disciplines from Computing Curricula 2005.

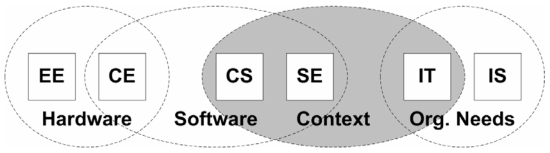


Figure 2. Disciplines from Computing Curriculum 2005, Emphasizing the Context for Software Design and Development.

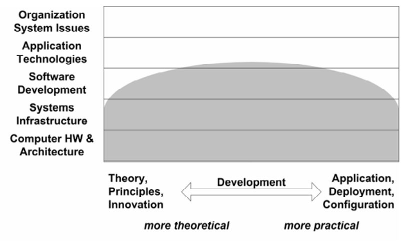
Consideration of the surrounding context appears in more traditional CS and IT programs as well — for example, as an HCI course in a CS major or an organizational impacts course in IT. The way to read Figure 2, then, is that each of the ovals frames the degree programs contained in it. Hardware is the key framing for electrical engineering, as are organizational needs for information systems. For the degree programs "in the middle", multiple framing factors exist and must be taken into consideration. Typically, though, one of those factors dominates. For computer science, this dominating factor is software, and for information technology, it is organizational needs. For software engineering, it is the context in which the software will be deployed. After all, the success of software engineering is measured not by the software we develop, but by how well that software addresses the contextual problem at hand.

At UC Irvine, we have created and instituted a new degree program that focuses on context: our new B.S. in Informatics. We define Informatics as the study of the design, application, use, and impact of information technology. Not surprisingly, then, at its core Informatics resembles software engineering, but this is augmented by many additional courses that broaden the focus (see Section 3) and by a different pedagogy that roots the delivery of the software engineering topics in context (see Section 4). To put this in perspective, we return to the ACM 2005 Computing Curricula document. It presents a series of diagrams that show the various areas of concern for different degree programs and further describes their focus as more theoretical or more practice-oriented. In Figure 3, we have reproduced those diagrams and added one diagram that shows the focus of Informatics. (The field of Electrical Engineering does not appear here, or in the source document, as EE addresses topics such as circuit theory that are outside the computing disciplines.) Informatics appears at the appropriate location in the continuum from hardware to information. Compared to a traditional software engineering degree, we note that Informatics has an equivalently strong focus on application technologies and software development, a less strong focus on systems infrastructure, and a stronger focus on organizational and systems issues. In Section 3, we further detail this focus in terms of the courses that we offer in the Informatics curriculum.

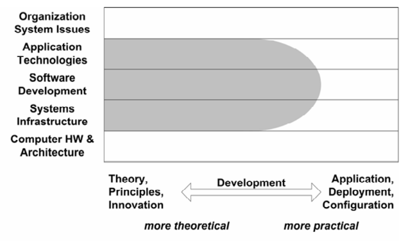
With the position of Informatics established, we now return to our definition of Informatics as the study of the design, application, use, and impact of information technology and derive the four principles that guided us in constructing our degree program. Each principle balances a traditional software engineering topic with an appropriate focus on context. In particular, we relate software and information, development and design, technical issues and social issues, and synthesis and analysis.

1. **Software and information.** Software is never a goal in and of itself. Instead, the real goal of software is always to manage some kind of information (by "manage," we broadly mean consume, produce, transform, visualize, store, recall, and similar activities). As such, software always provides its services as part of a larger system. Often its design requires careful consideration of how the software's management of information will affect the real world—not only when it fails but also when it behaves as desired. *Our first principle is that software engineering education must always place software in the context of the information it manages.*

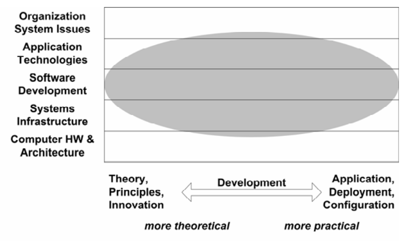
2. **Development and design.** Existing educational programs tend to focus strongly on teaching the overall software development process. The development process is an essential topic, but typical programs address each of the lifecycle phases equally. Furthermore, programs tend to frame design education in notations for documenting a design, typically with little design practice attached. This is shortsighted. At its core, software engineering is a design discipline, one in which the creative process of designing a solution is a central, yet very difficult, activity. The difficulty of design must be addressed explicitly by teaching of principles, learning from case studies, and extensive practice—not with notations alone. *Our second principle is that software engineering education must place development in the context of design; that is, it must treat software engineering as a design discipline.*


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

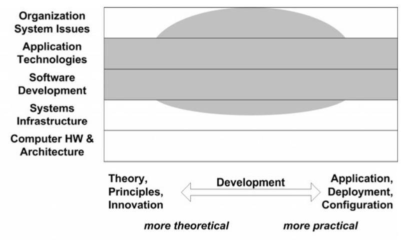
(a) Computer Engineering (CE)


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

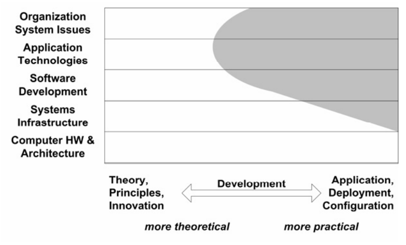
(b) Computer Science (CS)


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

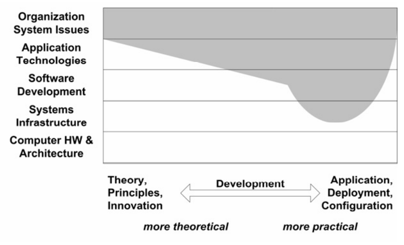
(c) Software Engineering (SE)


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

(d) Informatics


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

(e) Information Technology (IT)


Organization System Issues
Application Technologies
Software Development
Systems Infrastructure
Computer HW & Architecture
Theory, Principles, Innovation
Development
Application, Deployment, Configuration
more theoretical
more practical

(f) Information Systems (IS)

Figure 3. Areas of Concern for Various Degree Programs.

3. **Technical issues and social issues.** Software engineering is not just a technical discipline. While technical issues form a vital core, they should not be the only issues we consider. Software is always placed in social settings, and we must address those as an integral part of our teaching. This is not the same as just inserting a human-computer interaction course, though that is an important part of software engineering education. The need for educating students about social issues goes much farther and must include extensive treatment of topics such as organizational impact, cultural differences, and social responsibility (e.g., ethics, privacy, and security concerns). *Our third principle is that software engineering education must place technical solutions in the context of the social structures in which those solutions will operate.*

4. **Synthesis and analysis.** At the heart of software engineering lies our ability actually to construct software. Such construction, however, must be accompanied by careful analysis – both before and after construction. By this we do not mean only algorithm analysis, requirements gathering, or testing; we also include organizational assessments, feasibility studies, cost analyses, user studies, and so on. These kinds of analysis critically inform the software development process. *Our fourth principle is that software engineering education must place its focus on synthesis in the context of analysis.*

Following these principles, we designed the Informatics curriculum as described below. The result is an education that provides much broader perspective than traditional software engineering alone. We further note that any degree program wishing to address these principles must address them from all eight perspectives—software, information, development, design, technical issues, social issues, synthesis, analysis—and employ those perspectives pervasively throughout the entire program. The topics must be introduced and taught in an integrated manner so students will appreciate the breadth of software engineering from the start; otherwise, students will treat each topic as its own insular domain, missing out on the essential interconnectedness in the discipline.

3 Our Curriculum

The Informatics major is one of four majors offered in the Donald Bren School of Information and Computer Sciences: (1) Computer Science and Engineering, jointly with the Henry Samueli School of Engineering, (2) Computer Science, (3) Informatics, and (4) Information and Computer Science. The relationship among these majors is illustrated in Figure 4. We observe that the first three degree programs adhere to the topic divisions laid out in the previous section. The fourth degree, Information and Computer Science, is the original degree offered by the school. The school continues to offer this highly configurable major, since it serves students who want to sample courses across the spectrum (although it sacrifices some depth for the breadth it affords).

The major in Informatics is a four-year undergraduate bachelor of science degree [4]. Courses at UC Irvine are offered in three ten-week quarters per academic year. (A summer session is also offered, but summer attendance is not required. Students typically use the summer to catch up with courses they missed or to push ahead if they are on an accelerated schedule.)

Table 1 shows the entire four-year Informatics curriculum at a glance. Unlabeled courses are offered by faculty in the Department of Informatics, which is the academic home for the major. Courses typically taught by faculty in the Computer Science Department, the sister department to the Department of Informatics, are designated (cs). Courses offered outside of the Bren School of Information and Computer Sciences are designated (o) or (b), depending on whether the course is specifically required for the Informatics major or a course that students can freely choose to satisfy UC Irvine's

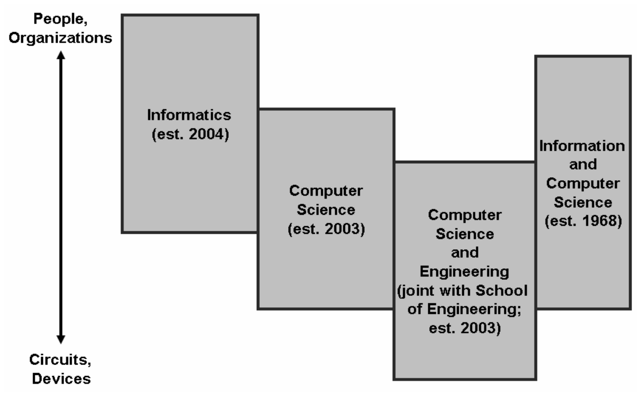

People, Organizations
Circuits, Devices
Informatics (est. 2004)
Computer Science (est. 2003)
Computer Science and Engineering (joint with School of Engineering; est. 2003)
Information and Computer Science (est. 1968)

Figure 4. Areas of Concern for UC Irvine Degree Programs in Computing.

breadth (general education) requirements. Finally, as we will discuss below, light grey shading indicates courses with significant treatment of software's broader context.

At the center of the first-year program is the year-long Informatics Core Course, a broad introduction to the field that carries a 50% higher unit load than the typical course at UC Irvine. The first two quarters introduce programming in Scheme and in Java[1] and include conventional first-year data structure and algorithms topics, but beyond that they also give students a background in the fundamentals of computing technology (the Internet and the Web, a functional overview of operating systems,

[1] Our choice of programming languages was driven by our overall goals for the programming portion of the year-long course. Beyond building students' programming skills, we want the material to be accessible to students without prior experience, to provide a design methodology rather than leaving students to flail at their code without direction, to emphasize high-level concepts like design and abstraction over low-level coding details (while still producing executable programs), and to foster students' appreciation for the variety of tools available and the criteria for selecting the most appropriate for a given task. We find that first-year students often feel swamped by the sheer volume of small details they must master. As novices, they have yet to develop any conceptual framework for organizing those details and deciding which are important in a given situation. When every asterisk, semicolon, or brace may carry crucial semantic information, confusion abounds. The syntactic and semantic simplicity of Scheme minimize this confusion, allowing an early concentration on data abstraction and program design (which in turn provides an excellent foundation for object-oriented programming and design). We also find that a two-language approach helps students distinguish between fundamental concepts and mere language artifacts, reduces their tendency to be uncritical fans of the first language they learn, and helps them avoid looking at every problem from the perspective of a single language's features and idioms. We cover Scheme in the first quarter and migrate to Java as a mainstream, industrial-strength language in the second quarter.

exposure to machine-level instructions and programming, bit-level and higher-level data representation) and opportunities (particularly via case studies) to approach broader, contextual issues such as human-computer interaction and the social and organizational impact of computing. The third quarter of the Core Course is an introduction to software engineering with a strong emphasis on design alternatives. Also in the third quarter, students take a half-course seminar that introduces them to the research interests and projects of the faculty in the Informatics Department. This comes relatively early in their program so they get an early exposure to the broad possibilities in the field (overcoming the impression some first-year students receive that computer science is only about programming).

In their first year, students also take the required university writing courses, a course in logic, a course in discrete math, and a Java-based course in intermediate data structures (at the level of balanced trees). This establishes some fundamental skills that they will build on in subsequent years.

Students who decide on Informatics during their first year, having enrolled in another introductory computer science course sequence, will generally be permitted to apply that sequence in place of the Informatics Core Course. While these students will not have the complete background we would prefer, we felt this was outweighed by the advantages of allowing a seamless transfer into the Informatics program.

The anchor of the second year is a three-quarter series in software engineering: a course in software methods and tools, one in requirements, and one in specification and quality engineering. Students also take a two-quarter sequence in programming languages (considering not only different programming paradigms and language implementation details but also domain-specific languages and scripting languages), a two-quarter sequence in human-computer interaction, a course in statistics, and the first quarter of a series on software design.

This year provides students with many practical tools and techniques, approaches they will use in their later work. But these courses emphasize the underlying concepts: what is the problem, what are the symptoms, what are the underlying issues, what is the spectrum of potential approaches to address the problem, what are the characteristics of those approaches, why might we choose one approach over another, and so on. Only after we introduce those concepts do we hone the practical skills with specific exercises using particular tools and methodologies. The result is that students, after the second year, have at their disposal a portfolio of foundational techniques and skills that further establish the discipline of Informatics.

The third year, aside from university breadth requirements and electives, includes a three-quarter series on the social and organizational impact of computing, culminating in a field-study project course. This series provides the broadest context, ranging from privacy and intellectual property through organizational issues and methodology for conducting field studies. The year also includes the second course on software design, a course on designing software architectures and distributed systems, and a first course on databases. This year is key to the curriculum in terms of design, examining it from two perspectives: social and technical. Each course introduces students to the issues involved from one perspective, but weaves in some from the other perspective to properly balance the presentation of the material.

Table 1. Required Curriculum for the UC Irvine B.S. in Informatics.

	Fall	*Winter*	*Spring*
First	Informatics Core	Informatics Core	Informatics Core
			Informatics Research Topics Seminar
	(o) Critical Reasoning	(o) Discrete Mathematics	(cs) Fundamental Data Structures
	(b) Writing	(b) Writing	(b) Writing
Second	(o) Statistics	Human-Computer Interaction	Project in Human-Computer Interaction and User Interfaces
	(cs) Concepts Programming Languages I	Concepts Programming Languages II	Software Design I
	Software Methods and Tools	Requirements Analysis & Engineering	Software Specification and Quality Engineering
	(b) Breadth	(b) Breadth	(b) Breadth
Third	Social Analysis of Computerization	Organizational Information Systems	Project Social and Organizational Impacts of Computing
	Software Design II	Software Architecture, Distributed Systems, and Interoperability	(cs) File and Database Management
	(b) Breadth	(b) Breadth	(b) Breadth
	(b) Breadth / Elective	(b) Breadth / Elective	(b) Breadth / Elective
Fourth	Senior Design Project	Senior Design Project	Senior Design Project
	(cs) Project in Database Management	(cs) Information Retrieval	Information Visualization
	Project Management	CSCW	(b) Breadth/ Elective
	(b) Breadth / Elective	(b) Breadth / Elective	(b) Breadth / Elective

The fourth year includes a three-quarter capstone course, a database project course, a project management course, a course on computer-supported collaborative work, and a pair of courses on information retrieval and visualization. The capstone course is a year-long senior design project in which student teams work with real clients to specify and implement a realistic system. Under the supervision of an interdisciplinary faculty team, students must bring together material from previous years (tools, skills, processes, ethnographic methods, design approaches, and many others) to complete their project successfully.

Returning to our principles, we have shaded in Table 1 the courses where context plays a significant role. Ideally, this would occur in every single course, but that would be unrealistic. Some courses are offered by another department, which does not always share our vision with respect to the importance of context, focusing instead on the traditional and theoretical aspects of computer science. Other courses present basic software engineering techniques; these courses are foundational and focus mainly on introducing the concepts behind the techniques and exemplifying their application. Nonetheless, most of our courses, from the first-year introductory course to the final capstone project, take a contextualized approach.

Examining our principles in more detail, we make the following observations:

1. **Software and information.** Information plays a critical role throughout the curriculum. It is the prime topic in the two-course database sequence and the information retrieval and visualization courses. But our coverage goes much beyond that. In the introductory course sequence, for example, we avoid the traditional programming-only approach in favor of one where projects are presented and addressed in terms of information needs (e.g., a project to enhance the design of a web store first involves considering the proper information flow and then examines the technical details of how this information flow is supported). The two software design courses are another example of courses in which information plays a critical role: The case studies and assignments focus on defining the information needs and then working those into software solutions.

2. **Development and design.** Some part of the curriculum has to introduce the development focus. We do this in the third course of the Informatics Core series and in the second-year software engineering series. Beyond that, however, design takes center stage throughout the curriculum. We paid particular attention to design in the introductory course. While students certainly are not asked to perform full-fledged design exercises, the materials are presented and exercised from a design point of view, which we deem critical to instill a sense of design right from the beginning of the major. A subtle but important example of how our design focus influences the courses and material we teach is the Concepts in Programming Languages II course. While a traditional course like this would focus on the construction and operation of compilers, our course is more about “little languages”. By this we mean that quite often one is forced to incorporate some kind of mini-language in a design solution, whether directly exposed to the users or internally processed.

Our course is about choosing the appropriate kind of language for a particular task, including situations where a solution requires some kind of mini-language designed and built for that task.

3. **Technical issues and social issues.** The three-quarter series on the social and organizational impact of computing, offered in the third year, is the primary place where students will learn about the social issues involved in software development. This series covers in detail topics such as privacy and intellectual property; social, ethical, and cultural differences and issues; and organizational issues and methodology for conducting field studies. That said, the material is truly put to the test in their senior design capstone course, where they will be designing and implementing actual systems that will have to pay attention to these factors. In addition, the first year Informatics Core course provides the students with a preview of these issues and the courses on human-computer interaction also pay attention to these factors even though these courses focus more on the actualities of user interface design.

4. **Synthesis and analysis.** Analysis returns in many forms throughout the curriculum, both with respect to the internal structure of a design or process and the externally visible properties of the software Analysis of the internal structure occurs, for example, in the design courses, which focus heavily on how to make informed tradeoffs. Analysis of the external manifestations of software occurs in the HCI course, where students learn how to perform user studies, and in the Social Analysis course, where students learn to recognize and analyze the effects of technology on individuals, organizations, and society.

Additionally, the Informatics program includes courses in logic, discrete mathematics, and statistics. The technical and mathematical underpinnings of the field form an essential component of the program; these skills are necessary for performing statistical analyses for user studies and certain tradeoff analyses of alternative design choices. Our choice, however, is not to promote additional math as a mechanism to raise "mathematical maturity" or to make the major "more rigorous". It is our belief that these more general benefits attributed to math can also be achieved in other ways, and in particular through building these skills in the domain in which they will be applied. In our case, this is through the study of the design, application, use, and impact of information technology. By framing software engineering through the consideration of context, the topic is sufficiently rich to challenge the students deeply and instill in them a strong sense of critical thinking.

4 Pedagogy

A program as diverse as Informatics must be taught in an interdisciplinary way by an interdisciplinary faculty. The Department of Informatics is composed of individual faculty members with diverse backgrounds, including law, programming languages, computer-supported cooperative work, human-computer interaction, psychology, anthropology, ubiquitous computing, and, indeed, software engineering. It is important to note that these faculty members not only engaged in the overall design of the major but also significantly shaped the pedagogical approach for delivering the individual courses.

We employ several best pedagogical practices throughout the curriculum, at varying levels of detail. At the highest level, we note that most of the curriculum is structured in course sequences rather than individual, disconnected courses. First, this brings continuity to the program, allowing us to treat important topics in depth while still supporting a gradual introduction to the more complex topics such as software design and information management. This is particularly important for those topics that require extensive practice complementing the lectures. With multiple courses, there is room to practice more and larger examples. A second reason for creating course sequences is that it allows us to integrate materials that should be taught together, particularly with respect to the principles we laid out in Section 2. Traditional courses are already typically full. Rather than simply creating a new, separate course to provide the additional context, we have combined, reorganized, and redistributed the traditional and newer material in an integrated manner over several courses. This is particularly evident in the Informatics Core series, which provides a balanced view of context that frames the traditional orientation of introductory computing courses around programming and data structures. It is present in the other course sequences as well. The first course in the software design sequence, for instance, first introduces students to general design process and principles (i.e., as involved in different kinds of disciplines) and only then makes the transition to apply this general knowledge to software. This is only possible because there are three courses in this sequence, allowing us to take a broader look at design before we address the specifics of notations, metrics, and such topics as architectural styles and design patterns.

A second high-level pedagogical practice we adopted pervasively was that of group work. Software engineering is not an individual activity; every software engineer interacts with people, whether it is peers (e.g., their software engineer colleagues) or others (e.g., customers, consultants, management). For this reason alone, it is necessary that students receive training in group work. But it is also a settled pedagogical principle that frequent group work provides a stimulating, engaging, supportive, and effective learning environment for students [5]. We begin in the first year with the use of pair programming, which we broaden in later years with projects involving larger groups of students. We ensure that students do not always work with the same set of other students, particularly in the first year. This helps provide them with a balanced experience and also promotes a sense of community as the students move through the program. We also note that we provide the students with the tools and approaches necessary to perform their work effectively in groups. The second year in particular provides them with methods and tools for configuration management, bug tracking, modeling, process management, requirements solicitation and management, and testing, analysis, and so on. The expectation is that they will use those in subsequent years as needed, particularly in the final year-long project course. We also note the project management course that is slotted at the beginning of the fourth year, but can be taken earlier in the program of study (as desired by individual students and depending on their progress).

Case studies are used frequently to seed the courses and provide students with real examples that set an appropriate context for the material being taught. The course on social analysis of computerization, for instance, uses quite a few examples of "computing gone wrong"; not in the traditional sense of a crashing program, but in the sense of mismatches between technology and its actual use. Similarly, the software design course sequence uses many real-life examples to illustrate designs, both good and bad, in terms of the code structure they prescribe. Some courses ask students to find their own case studies, for instance by asking them to choose a software application and perform a detailed analysis of how effective it is in use or how a particular user interface is perceived by a target group of users. When a full, realistic, detailed case study is not feasible, we attempt to provide alternative ways of approaching real-life situations as realistically as possible. For example, we use a tool that simulates the software process to allow students to visualize and practice different approaches to the software development life cycle (e.g., waterfall, incremental, XP) [6].

Each year ends with a capstone course that brings together the materials taught in that year. As mentioned previously, the senior year is anchored by a year-long capstone course, the Senior Design Project. In it, students work with real clients to specify, design, and implement a solution to a particular IT problem that the client faces. Past experience with a quarter-long project course shows that students do high-quality work and the students enthusiastically cite the course as one of the most influential in their future careers. The quarter-long course, however, is too hurried and does not offer much time for iteration and full delivery of the results. (Though we have seen some remarkable successes in which systems created by a student team are readily adopted and "go live" shortly after delivery, students must be able to sleep, too!) The year-long version brings stability in this regard and also allows us to tackle systems of larger scale and broader variety, especially because students will have been prepared much more thoroughly with the skills necessary to successfully undertake their projects. Moreover, it provides an opportunity to exercise requirements gathering (which is hardly done now in the one-quarter course) as well as UI design and evaluation, specification changes (which are inevitable over the course of nine months), final packaging, deployment, and installation.

Our overall approach and courses incorporate many different learning theories (learning by doing [7], situated learning [8], learning through failure [9], Keller's ARCS [10], and so on). This is an artifact of having been able to design many of the courses from scratch, a luxury we realize one does not always have. This also allowed us to insert, in the spring of year one, a seminar course in which the students meet each of the faculty members and hear about their research. This is important for building community and putting a face on the program, but it also better enables the students to approach the faculty later if they want to work on a research project. The program is designed to allow room for semi-independent research projects, a minor in another field, or free elective classes.

5 Reflection

The first students entered the Informatics program in Fall 2004; as of this writing, they are now sophomores (about 20 students) and a second cohort is completing its first year (about 30 students). We have instituted an ongoing, detailed, long-term evaluation process to determine the effectiveness, strengths, and weaknesses of our approach; clearly, just two years with students who have not yet completed the program is insufficient to draw firm conclusions. As part of this process, we are closely monitoring the performance of the Informatics students as compared to students in the other computing majors in the Bren School. We also gather information from a significant amount of informal contact. Here, we provide some of the data that we have collected, some of our anecdotal experiences, and our impressions of the program structure so far.

5.1 Student Performance

Table 2 presents the data we have collected to date on the GPA per quarter for students who entered any of the Bren School of ICS computing majors in Fall 2004 or Fall 2005.[2] This represents the two years that the Informatics major has been in existence. Overall, we note that the GPA of Informatics students is somewhat lower than that of students in the other majors, but that the best students in any of the majors are comparable. The lower GPA can be attributed to many factors and this comparison is like comparing apples and oranges. Students in different majors take different courses with different instructors; Informatics students in particular take some courses in their first and second years that are third- or fourth-year courses for the other students; students have different characteristics (see below); and so on. Monitoring students' performance also ensures that the major does not pose significant bottlenecks that impede students' progress.

Some of the courses that Informatics students take are shared with the other majors. In these courses (data not shown here) we see some polarization: the best Informatics students perform as well or even better than the other students, but other students are a bit worse in their performance. That is, the traditional normal distribution does not quite apply to our students in these shared courses. We have two hypotheses: (1) the shared courses are junior-level courses for the other students, so they have more practice and experience than the typical Informatics student in the same course, (2) more than students in the other majors, Informatics students fall into two distinct camps – those who like to program and those who do not. This last factor is interesting: the Informatics major attracts a higher percentage of students who do not wish to become traditional computer scientists eventually but rather wish to enter careers in interaction

[2] Data are collected only for students who have consented to be part of an official study that is sanctioned by the UC Irvine Institutional Review Board (IRB). Our reporting here is therefore constrained to just those students and limits itself to non-identifying data that can be shared under the current IRB guidelines.

design, game design, management, consulting, film production, and other careers that require very little hands-on production of code and instead require a deep understanding of overall design issues. These students tolerate the shared Programming Languages and Software Methods and Tools courses, and we believe that a solid grounding in the technical fundamentals is essential for those who will be managing or working closely with hands-on programmers, but they are seldom the students who excel in those courses. By the same token, we expect them to perform well in courses that are oriented towards design issues.

Table 2. GPA per Quarter for Students in each of the Four Computing Majors of the Bren School of ICS.

Major	*Qtr*	*Mean*	*Median*	*SD*	*Min*	*Max*
CS	F05	2.95	3.09	0.95	0.00	4.00
	F04	3.40	3.57	0.55	1.91	4.00
	W05	3.14	3.33	0.59	1.86	4.00
	S05	3.18	3.30	0.56	1.78	4.00
CS&E	F05	2.85	3.05	0.91	0.00	3.77
	F04	3.15	3.42	0.73	1.48	3.98
	W05	3.13	3.09	0.46	2.45	3.92
	S05	2.92	2.95	0.64	1.75	3.76
ICS	F05	2.84	2.98	0.83	0.33	4.00
	F04	3.03	3.10	0.76	1.33	4.00
	W05	3.04	3.11	0.71	0.83	4.00
	S05	3.01	3.11	0.79	0.00	4.00
Informatics	F05	2.88	3.07	0.96	0.26	4.00
	F04	3.22	3.51	0.66	1.91	3.91
	W05	2.93	3.13	0.86	0.29	3.93
	S05	2.74	3.33	1.22	0.00	3.91

We also note that the gender distribution is quite different for the different majors. Informatics attracts a significantly higher percentage of female students than the other majors. At present, about 25% of the Informatics students are female, a percentage about twice as high as the other three majors. We attribute the difference to the focus of Informatics. Because of its integral treatment of context—particularly people and organizations—and because of they way we have structured even the introductory courses to integrate this focus on context, the major is more accessible, has a clearer structure, and has much more grounding in realistic situations. These are precisely the factors that have been identified as critical for achieving and maintaining a gender balance in computing degrees [11].

5.2 Anecdotal Experience

Our anecdotal experience reveals great enthusiasm among the current Informatics students. They enjoy the challenge of participating in a new program that is at the frontier of computer science education and they spread the word to students in the other majors. They have even organized an Informatics Student Association with the mission of spreading awareness and information about Informatics, both on campus and off, to high school students and industry.

We have made some adjustments to the program as circumstances warranted. In the first year of the program, we noticed that more students than expected left the program. We attributed this in part to a mismatch in expectations; our initial recruiting materials and presentations, in emphasizing the context to distinguish Informatics from the more traditional degrees, may have left the impression that the Informatics program did not involve a significant amount of programming. But it does, as we have described above, and we have edited our materials to reflect this more accurately with the result that our second-year attrition rate to date is lower. Another factor contributing to attrition is that each Informatics course is offered exactly once a year, while courses in the other, larger majors are typically offered more often. A student who does not pass a required Informatics course may need to wait an entire year to repeat it and to take other courses for which it is a prerequisite. Some students in this situation have switched to the Information and Computer Science major when they failed one of the critical courses. We hope that, in the future, the size of the program and of our faculty will grow, enabling us to offer these critical courses more frequently.

We have also observed a high degree of interest in Informatics from juniors and seniors in other computing majors, higher than from first- and second-year students. We hypothesize here that "younger" students have a more limited view of the alternative ways to study computing and tend to opt for the majors with the more conventional, familiar names. As they progress through those programs, they learn more about their preferences and potential future careers. At that point they often recognize the value of Informatics. This parallels our recruiting experiences with local high schools: prospective students know the Computer Science name and don't look any further. On the other hand, when a prospective student's parents are in the computing industry themselves, often they appreciate the value of the Informatics approach. As a School, we are updating our promotional materials to describe and distinguish the majors and to help students with their choice.

5.3 Program Structure

Although the Informatics program is still quite young, we re-evaluate it periodically. At this point, we are contemplating a few changes to the ordering of the courses:

- *Move the database course series one year earlier in the program.* With the prevalence of databases and their likely inclusion in many of the senior design projects, we feel that an earlier exposure will be helpful.

- *Move the project management course into the junior year.* Learning this material before starting the senior design project should help students carry out their project more effectively.

- *Delay the software design course sequence by one quarter.* Besides making room for the changes listed above, this rearrangement will allow students to consider database issues in their study of design.

Continued experience in teaching the program undoubtedly will bring additional fine-tuning, but we are confident that the Informatics program, especially with these changes, presents a well-balanced education in the design, application, use, and impact of information technology.

6 Challenges

In creating our Informatics curriculum and continuing our efforts to improve it, it is clear that the field of software engineering still faces many challenges on the way to becoming a mature discipline. Here, we discuss the four that have the greatest impact on our curriculum; we believe they apply to other software engineering degrees as well, regardless of how they are framed.

1. *How to balance teaching to students who like programming and students who do not.* Our discipline must nurture students of both varieties; not every software engineer will be a coder. In fact, with the growth of outsourcing, we should examine our curricula carefully to ensure that we do not educate “just programmers”. While most software engineering degree programs do offer some kind of broader perspective, perhaps it is time to engage in the discussion of how broad this perspective should be and how large a role programming should play. One could argue that programming is a fundamental principle underlying all of software engineering, but one could equally argue that we should be able to train designers (in the broadest sense of the word) without their having to be master coders (since, for example, building architects are not necessarily trained, or even competent, in the masonry or carpentry necessary to construct their designs). This discussion is merely beginning right now, and our Informatics degree simply occupies one spot in the space of possible solutions. We have not abandoned or curtailed programming, but we have placed a much heavier emphasis on design and context to balance programming with other skills.

2. *How to teach design.* To date, our focus as a discipline has centered mainly on notations for capturing designs. While these notations are tremendously useful, their focus on the end product limits their usefulness during the creative, exploratory, iterative design process, which we want our students to

learn. For our Informatics degree, we have created a course series that introduces a novel theoretical perspective on design in general before focusing on software design. This helps set the appropriate expectations, brings the treatment of software design in line with other disciplines, and changes the focus from notations to theory, examples, case studies, and extensive practice. A theoretical framework by itself is not sufficient, however. We must invest significant effort in collecting examples of good and bad designs, creating new design environments that focus on supporting the creative process rather than the documentation process, and collectively figuring out ways to understand what works and does not work in design education.

3. *How to incorporate context in the educational experience.* Our Informatics major has chosen to interpret software's context in terms of information, design, social issues, and analysis; it addresses the context with specific courses and a concerted effort to provide a broadly balanced view throughout. This represents but one approach to incorporating context. The Informatics major at Indiana University provides another example through the use of focused plans of study in other domains, to balance the more technical material related to computer science [12]. The introductory courses at Georgia Tech use media computation (graphics and audio) to provide a more contextualized approach to the introductory topics in the major [13]. Many approaches and experiences are emerging; an examination and discussion of their relative merits is much needed.

4. *How to develop students' appreciation for issues of complexity and scale.* It is well known that the projects one can feasibly undertake in an educational setting are significantly smaller in scope than those undertaken in industry. Issues that crop up in these larger projects may not arise in the smaller projects. We must find ways to expose students to such issues nonetheless. The literature mentions many techniques for addressing these issues (e.g., 20 dirty tricks [14], maintenance projects that continue from year to year [15], purposely handing out unclear requirements [16]), but these tend to focus on specific symptoms. They illustrate one or two visible results of what may go wrong, but do not comprehensively discuss and explore issues of scale. The Informatics major includes a year-long design project in which students undertake a project for a real customer. It also uses project simulation software for "hands-on" practice with larger, hypothetical situations in a safe, virtual setting [6]. We view all of these as beginning to address complexity and scale, but we believe that much more work is needed to create many more educational approaches that do so in a principled way.

Answers to these questions are at the heart of a high-quality SE education. The lessons we learn as we answer them will benefit the community at large and the sooner the community sorts them out, the better.

7 Conclusions

We have presented the new Informatics major at UC Irvine, the result of our attempt to create a new way of framing software engineering education,. What distinguishes our approach is a framing in context: we integrate coverage of software and information, development and design, technical issues and social issues, and synthesis and analysis. We believe that in this way, we will train well-rounded software engineers who have flexibility in their careers and a deep understanding of the issues involved in creating new information technology.

As technology advances and our knowledge increases, a complete education in computing no longer fits into a single four-year degree program. Software engineering is a leading candidate for “splitting off” into a separate degree (much as mechanical, electrical, and other engineering disciplines became programs separate from general engineering). Software engineering is criticized in some quarters as being more vocationally oriented than the usual academic discipline. An Informatics approach, integrating software creation with context drawn from many traditional academic fields, may go a long way towards answering these critics and developing software engineering as a mature discipline.

We are not alone in attempting to revamp software engineering education. Jazayeri describes a similar effort at the University of Lugano [17]. Earlier we mentioned the Informatics program at Indiana University [12]; other similar efforts are underway [18,19,20,21,22]. As software engineering educators, we face many challenges — not only the pedagogical ones listed above but also the challenges of rapid technological change, global increase in demand for information technology, and the development of a global IT workforce. To prepare our students for productive careers, we need more support, experimentation, evaluation, and discussion of alternatives such as these.

8 ACKNOWLEDGMENTS

The Informatics major at UCI is sponsored in part by the Fund for the Improvement of Postsecondary Education (FIPSE), U.S. Department of Education.

References

1. ACM, AIS, and IEEE-CS Joint Task Force for Computing Curricula 2005, Computing Curricula 2005, http://www.acm.org/education/curricula.html
2. McMaster University, Department of Computing and Software, http://www.cas.mcmaster.ca/cas/
3. IEEE-CS and ACM Joint Task Force on Computing Curricula, Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering (A Volume of the Computing Curricula Series), 2004, http://sites.computer.org/ccse/

4. University of California, Irvine, Department of Informatics, http://www.ics.uci.edu/informatics/ugrad/
5. Smith, *Teamwork and Project Management*, McGraw-Hill, 2004.
6. Oh Navarro and van der Hoek, *Design and Evaluation of an Educational Software Process Simulation Environment and Associated Model*, Eighteenth Conference on Software Engineering Education & Training, February 2005, pages 25–32
7. Schank and Cleary, *Engines for Education*. 1995, Hillsdale, NJ, USA: Lawrence Erlbaum Associates, Inc.
8. Brown, Collins, and Duguid, *Situated Cognition and the Culture of Learning*. Educational Researcher, 1989. 18(1): pages 32–42
9. Schank, *Virtual Learning*. 1997, New York, NY, USA: McGraw-Hill
10. Keller, and Suzuki, *Use of the ARCS Motivation Model in Courseware Design*, in *Instructional Designs for Microcomputer Courseware*, D.H. Jonassen, Editor. 1988, Lawrence Erlbaum: Hillsdale, NJ, USA
11. Margolis and Fischer, *Unlocking the Clubhouse: Women in Computing*, Cambridge, MIT Press, 2001
12. Indiana University, School of Informatics, http://www.informatics.indiana.edu
13. Guzdial, *Introduction to computing and programming with Python: A Multimedia Approach*, Prentice-Hall, 2004
14. Gehrke, et al., *Reporting about Industrial Strength Software Engineering Courses for Undergraduates*, Proceedings of the 24th International Conference on Software Engineering. 2002, pages 395–405
15. Sebern, *The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment*, Proceedings of the 15th Conference on Software Engineering and Training, 2002, pages 118–127
16. Daniels, Faulkner, and Newman, *Open Ended Group Projects, Motivating Students and Preparing them for the 'Real World'*, Proceedings of the Fifteenth Conference on Software Engineering Education and Training, 2002, pages 115–126
17. Jazayeri, *Education of a Software Engineer*, keynote at the Automated Software Engineering Conference, 2004
18. University of Washington, Information School, http://www.ischool.washington.edu
19. York College of Pennsylvania, http://www.ycp.edu/academics
20. Montclair State University, Department of Computer Science, http://cs.montclair.edu/undergraduate.html
21. Rochester Institute for Technology, Department of Software Engineering, http://www.se.rit.edu/degrees.html
22. Milwaukee School of Engineering, B.S. in Software Engineering, http://www.msoe.edu/eecs/se/