

# Teaching Summary

R. E. K. Stirewalt

My teaching philosophy has evolved over many years and has been heavily influenced by my experience in industry. It can be summed up in three points: First, whereas software is increasingly finding its way into every aspect of life, and whereas much of this software is written by engineers with only an undergraduate degree, our coverage of topics must be rigorous, emphasizing depth over breadth, and we must demand high standards of quality in student work. Second, good curricula exhibit conceptual integrity, which means that the required courses are designed and interconnected based on a clear and explicit vision of where the faculty thinks the field is headed. Finally, successful research in my own field (software engineering) is synergistically linked to education.

With a background in software engineering, I tend to teach the courses that emphasize rigorous development and quality. In fact, since joining the Department, I have taught these courses at every level, e.g., introductory programming (CSE 232), object-oriented design (CSE 370), software engineering (CSE 470), and formal methods (CSE 814). While it is relatively easy to be rigorous and to demand quality in a programming course, it is difficult to achieve the same goals in a course that is primarily concerned with design methods and heuristics, such as senior-level software engineering. I realized this problem early and did a lot of work to overcome it. This work culminated in a new course (CSE 370) and a different approach to teaching and evaluating skills in design, based on intensive laboratory exercises and cooperative learning during lectures (**Section 1**).

I have also worked to improve the conceptual integrity of our undergraduate curriculum. When I arrived, the composition of faculty was slowly changing from an emphasis in hardware and computer architecture to an emphasis in software, networking, and machine learning. However, the curriculum had not been updated to reflect these changes, and this was most evident in the area of software. I served on the curriculum committee as one of the principal architects in the new curriculum, which now has a much greater emphasis on software design and is thus more in line with the composition of the department (**Section 2**).

Finally, my research is concerned with the development, maintenance, and verification of high-assurance software. Much of this research yields new engineering models, methods, and notations—most of which represent a paradigm shift over the analogous models and methods currently used to develop software. Paradigm shifts involve a tremendous intellectual investment, and often an old paradigm can be replaced only by attrition [Kuh62]. Consequently, practitioners are slow to adopt new methods, and many never will. Thus, for this kind of research to transition quickly into practice, new methods and techniques must be worked into the undergraduate curriculum. A major obstacle to this integration is a lack of well-designed educational resources. I have addressed this problem with respect to various needs in object-oriented design, formal methods, and also writing and critical analysis (**Section 3**).

# 1 Contribution: Teaching design

The body of knowledge referred to as *software design* comprises best practices and heuristic methods for designing large systems subject to desirable properties, such as maintainability, extensibility, or separation of concerns. Such topics are difficult to teach because best practices and heuristic methods tend to be *metaphorical* rather than analytical. Fundamentally, metaphorical knowledge only has meaning in the context of experience [Say41]. Consider, for example, the design patterns of [GHJV95], each of which is expressed using a combination of scenarios, pictorial diagrams, and code schemas. Such a pattern, expressed in this form, is an expression of experience, and a designer with experience in object-oriented design will recognize it as such and be able to apply the pattern in a new context.<sup>1</sup> However, to an inexperienced designer, the pattern represents nothing but words and pictures, which can be memorized and regurgitated, but which will likely never spring to mind in a design context that could exploit it. Thus, the fundamental obstacle to achieving the objectives of a software-design course is to quickly and effectively give students a minimal base of experience necessary for them to comprehend the metaphors that constitute the meat of the course.

**Challenges to teaching design** The standard approach to teaching a design course is to assign a large group project, in which students have wide latitude to make design decisions, followed by a critical evaluation of their designs, usually near the end of a semester. This approach suffers from several efficiency problems, which I have worked to overcome. First, to formulate a project that will impart the experiences necessary for students to achieve the course objectives requires an inordinate amount of instructor time at the beginning of a semester. Second, during the semester, teams will need significant feedback on prototype designs, and often each team will make the same kinds of mistakes but at different times; thus the instructor will find himself repeatedly teaching the same content multiple times to small subsets of the students in the course. One way to address this problem is to break up the project so that students first produce a paper design, which is critically evaluated and refined before being implemented. The problem with this approach is that until the students have taken their design to implementation, they will not yet have sufficient experience to apply the metaphorical knowledge required to produce a good design. Finally, at the end of the semester, the instructor must evaluate a large set of large projects, each of which might involve thousands of lines of code accompanied by a 20-50 page design document.

This approach inherently limits the effectiveness of instruction because the most useful instruction tends to occur in small group meetings that take place outside of class, usually during an instructor's office hours. To overcome these obstacles, I developed a new software-design course (CSE 370) that incorporates two non-traditional approaches to teaching design. First, to address the student-experience problem, the course is run *bottom up*, starting from implementation and working slowly toward design and then specification. Second, in order to maximize the effectiveness of instruction during class time, the course employs a form of collaborative learning based on in-class design exercises that are worked on in teams. My experience in teaching this course now three times suggests that these techniques not only solve the efficiency problems but also increase student retention of these very abstract concepts.

---

<sup>1</sup>Contrast this metaphorical knowledge with other traditional concepts, such as a finite automaton, a priority queue, or a B-tree, each of which enjoys a formal (mathematical) meaning irrespective of the design context to which it might be applied.

**A bottom-up approach to teaching design** To impart the relevant experience, I designed an intensive regimen of laboratory exercises, each of which requires the student to add a new feature to an existing corpus of software, specifically a toolkit of graphical user-interface classes and a toolkit of distributed-computing abstractions. To add the new feature, the student must first understand and then redesign (or refactor) the existing classes and collaborations, which I designed for the purpose of illustrating the complexities that arise in large software. Each of these lab exercises gives students a basis in experience for understanding the lecture material, which presents a pattern or solution to address the complexity that was motivated in the lab. Thus, unlike in traditional design courses, which begin with a requirements specification followed by design and implementation, CSE 370 begins with directed implementation and maintenance tasks, which motivate design strategies and techniques.

**Collaborative learning to maximize instructor effectiveness** According to the bottom-up approach, each lecture addresses a problem, which is motivated in the lab, and culminates in a design pattern, strategy, or method that allows the problem to be solved systematically prior to implementation. Of course, applying these methods takes practice and requires critical instructor feedback. This need for personalized feedback will quickly exhaust precious instructor time if it cannot be handled during class, but it is clearly not possible to give personalized feedback to 50 students in an hour and a half. To address this obstacle, I designed CSE 370 to use *collaborative learning* techniques to provide personalized feedback in a way that reaches all of the students at once.

Following the introduction of each new pattern, strategy, or method, I assign an in-class exercise, which I ask students to work on, first by themselves and then in pairs, to formulate a solution. Working in pairs has two benefits: First, each student is forced to actually apply the technique rather than just copying it down. Second, and more important, a student's partner can often provide sufficient feedback on simple misconceptions, e.g., issues of programming-language syntax or misunderstanding what the exercise asks for. Consequently, many common misunderstandings are clarified by the students themselves, leaving the instructor to deal with the fundamental misunderstandings, of which there tend to be a very small number. I have found that the use of this technique dramatically increases the bandwidth of discussion and the overall retention of these concepts.

## 2 Contribution: Design of the undergraduate curriculum

Good curricula should exhibit conceptual integrity, but given the pace of change in our field, this goal is very difficult to achieve. When I arrived, the curriculum was more of a computer-engineering than a computer-science program even though our new faculty hires were increasingly software people. Since arriving, I have worked on our curriculum to discover and remedy discontinuities in the program, and I have provided intellectual leadership to transition our curriculum from a hardware-oriented program to a software-oriented program.

**What does object-oriented programming mean?** The first course I taught at MSU was the senior software-engineering course, which listed *object-oriented programming* (OOP) as a prerequisite skill. In our curriculum this skill is covered in the second introductory programming course (CSE 232). However, when teaching the software-engineering course, I noticed that students had no experience with the use of class inheritance and polymorphism, which are essential to OOP.

I then spent a year teaching CSE 232 and learned that the term was being used rather loosely to refer to the use of C++ classes to define *abstract data types*, which is not the same as OOP. This experience led me to develop a new course on object-oriented programming and design (CSE 370), and this course is now the bridge between our introductory programming courses and our senior software-engineering course.<sup>2</sup>

The root cause of the discontinuity is a familiar problem where the same term has different meanings to different people. It seemed clear to me that this terminology problem might be pervasive in our curriculum. I therefore began to precisely document the contents of my courses in terms of fine-grained *cognitive objectives* [Sti76]<sup>3</sup>, which document: (1) What is the learner to DO after instruction? (2) Under what conditions? and (3) What is the acceptable performance? In addition to using this technique for my own courses, I was able to get other faculty to do the same for their courses, and we discovered further discontinuities, which were subsequently fixed. I believe this effort has led to a significant improvement in the quality of our program, and the discussions that resulted from this work have helped to improve the conceptual integrity of the program.

**Defining and applying CQI** In addition to these benefits, fine-grained objectives are also the key to instituting continuous quality improvement (CQI) into our curriculum. Briefly, CQI is a principle by which a process can be incrementally improved through measurement of outcomes and the identification of root causes of failure. We are required by ABET to institute a CQI process in our curriculum prior to accreditation. I have taken the lead in this regard by employing these techniques as a pilot project in CSE 470, CSE 370, and CSE 232. Using these objectives, I then designed tests, homework assignments, and other assessment materials to test specific objectives; I also recorded the results of student performance on specific questions so that I could accurately measure student achievement on particular fine-grained objectives. The results of this work will be used not only in my course, but also in other courses in our curriculum. In fact, several other faculty have used my results in their own courses.

### 3 Contribution: Developing educational resources

Finally, I have spent a significant amount of time developing educational resources, including:

1. course modules,
2. self-contained handouts and essays on topics that are not yet adequately covered in text books,
3. laboratory exercises, including extensive libraries of software to support these exercises, and
4. design projects.

Some of the more notable course modules include lectures on technical writing and critical analysis, which I use in conjunction with graduate courses and seminars. Some of the more notable handouts

---

<sup>2</sup>I also made significant changes to the programming course (CSE 232), including adding more rigorous and difficult projects and adding modules that cover the use of specifications and contracts in designing reusable ADTs.

<sup>3</sup>In the Summer of 1999, I attended the NSF-sponsored Science and Engineering Education Scholars Program (SEESP) at the UW Madison. There, Professor James Stice (UT Austin) spoke about designing courses around measurable cognitive objectives as the best thing that a faculty can do to improve his or her teaching.

describe topics such as configuration management, role-based design, and object-oriented frameworks. I've used these resources in both graduate and undergraduate courses, as well as for training new students who join our research lab. The laboratory exercises for CSE 370 involve a large corpus of user-interface and distributed-computing software, which I developed. Finally, I have invested a significant amount of energy developing design projects for CSE 232, CSE 370, and CSE 470, and I share these resources with colleagues at other universities.

## References

- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.
- [Kuh62] T. S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962.
- [Say41] D. Sayers. *The Mind of the Maker*. HarperCollins, 1941.
- [Sti76] J. E. Stice. A first step toward improved teaching. *Engineering Education*, 66(5), February 1976.