An Experience Report on the Design and Delivery of Two New Software Design Courses

Alex Baker and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3440 U.S.A.
+1 949 824 6326
{abaker, andre}@ics.uci.edu

ABSTRACT

In this paper, we report on our experience in designing and delivering two new software design courses in the Informatics major at UC Irvine. When the major was created in 2004, it explicitly contained slots for two software design courses to be created from the ground up. The authors led this effort, focusing one course on the topic of *system design* and one course on the topic of *implementation design*. We discuss the philosophy and pedagogy behind the courses, present key class activities, and reflect on having offered each course twice over the past two years.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – Curriculum; D.2.2 [**Software Engineering**]: Design Tools and Techniques - Object-Oriented Design Methods.

General Terms

Design

Keywords

Software design, system design, implementation design, software design studio, general design thinking

1. INTRODUCTION

In Fall 2004, the University of California, Irvine inducted the first class of students into its new Informatics major, which studies the design, use, application, and impact of information technology (an overview of the major can be found here [1]). The major is structured around a core of courses in software engineering, databases, programming languages, computer-supported collaborative work, human-computer interaction, and social and organizational impact of information technology. Design is a theme throughout, which is why it was decided to include two full courses on software design in the curriculum (UC Irvine follows the quarter system, meaning that each course involves ten weeks of instruction).

The topic of software design is addressed in several courses in various existing majors at UC Irvine (i.e., Information and Computer Science, Computer Science, and Computer Science & Engineering), but the total amount of material would be insufficient to fill twenty weeks of lectures. Moreover, the opportunity to take a step back and consider what ideally should be taught in a pair of software design courses was appealing. Hence, the authors led an effort to design the two courses from the ground up.

Curriculum-wise, the first course is offered at the end of the sophomore year, after students have taken a general software engineering course, two HCI courses, two programming language courses, and a requirements engineering course. The second course is offered immediately following, namely at the beginning of the junior year, and sets the stage for an advanced software architecture course as well as, in the senior year, a nine-month capstone project involving an industrial partner. As such, we could count on a relative level of proficiency in a number of core areas (e.g., programming, design of user interfaces, role of software engineering), but also needed to prepare the students for the subsequent courses and their careers afterward by providing them with relevant skills, knowledge, and hands-on abilities.

In this paper, we describe our philosophy and pedagogy in designing and delivering the two courses, present key class activities that illustrate how students participated and learned in the courses, and reflect on having taught the courses twice over the past two years.

2. PHILOSOPHY AND PEDAGOGY

In designing these courses, we examined the existing literature in software engineering education, educational standards (e.g., IEEE and ACM Curricular Guidelines [2], SWEBOK [3]), and courses in software design at other institutions that we knew of or otherwise found by searching the Internet. We learned that an overarching focus is on the design *artifact*, and especially the patterns and notations in which the design artifact is to be captured. Much less is said about the design *process* via which a design is ultimately created. Adoption of a studio approach to software design [4, 5] was most promising in this regard. Studios draw their inspiration from other design fields, where frequent and open critique is a key technique in honing students' design skills [6].

We wanted to provide a broader perspective on design, as inspired by the general design literature, as well as a deeper examination of the more traditional topics, as afforded by the twenty weeks of instruction available to us. Guiding the philosophy and pedagogy of the two courses, then, are the following four key principles.

Provide a General Design Perspective: The authors have spent time previously in studying design in general, as it exists in fields other than software engineering, and in relating the resulting lessons back to software design [7]. We wanted to cast the courses in this perspective, framing software design relative to these other fields, illustrating how software presents a unique kind of design problem, but at the same time showing how it is not wholly unlike the kinds of design problems faced in other fields. People have a natural ability to design, and an intuitive understanding of the role of design in other fields [8]. We believe that leveraging this resource makes software design more accessible and allows students to more easily leverage their inherent problem-solving skills.

Encourage Design-Minded Thinking: Designers exhibit a broad variety of behaviors when faced with a given design problem. It is well-known in other fields that designers often work opportunistically, proceeding with a given direction as long as possible. More experienced designers also know when and how to generate alternatives, as well as how to properly evaluate them. They will gather relevant (domain) knowledge to back up decisions, or readjust a problem definition when it does not enable them to design the right solution. None of these ideas are explicitly taught in software design courses, yet the portfolio of approaches that a designer has to enact such behavior is critical to their success in designing. We wanted our courses to explicitly discuss and promote these "designerly" behaviors [8].

Separate System Design and Implementation Design: Software engineering courses typically let students practice a design largely on their own. A requirements document is provided and a student must hand in a design document several weeks later, which is then graded. Design, however, is much broader, and, in the real world, customers' requirements are imprecise and incomplete, requiring a designer to augment and carefully interpret them. Designers are therefore often involved in actually setting requirements, working in teams representing a range of different stakeholders, and generally framing a design problem and its associated solution, sometimes from the highest levels to the smallest details, with important design decisions made throughout. Our courses therefore separate *system design* from *implementation design*. The first course centers on system design, which involves deciding what functions a software system should provide and how it should interact with the world. The second course centers on implementation design, introducing the more traditional subjects of designing the internal structures that facilitate a system's implementation and its maintenance. This division, although it admittedly sets a boundary that in practice is less stringent, allowed us to focus our lectures and topics, and allowed students to more easily relate their activities to non-software design fields.

Balance Theory and Practice: Any designer's ability to generate quality software systems is highly influenced by their experiences amassed over the years [9]. While including a generous portion of hands-on practice, often in a studio setting, we also structured our courses to provide a strong theoretical foundation through which students can contextualize their practical work. We believe that this helps students assimilate their experiences in class while at the same time laying a basis for long-term learning, to be applied when the students become practicing software engineers.

In short, we strove to teach students to think critically about software design problems, spanning from early decisions during system design to later yet equally important decisions made in implementation design. We particularly wished to provide a blend of interdisciplinary theoretical foundations and practical experiences to allow students to hone a designerly mode of thinking.

After offering both classes a first time, we made several improvements in how we brought this described philosophy and pedagogy to the students, the results of which are described below.

3. SYSTEM DESIGN COURSE

The first course introduces students to a variety of general design concepts and theory, and explores the ways in which software can be designed to meet real-world needs. The course not only teaches students about software. It also broadly promotes effective design methods, problem solving skills, and creative thinking. Here, we explain several approaches that we used to teach these lessons.

3.1 Non-Software Design Activities

In perhaps our most radical departure from traditional approaches, during the first three lectures, students use a variety of materials to design non-software products, such as trophies, bridges, and towers. We do not even broach the subject of software, simply opening the first lecture with such a non-software design exercise. We have three objectives in using this approach:

1. *To get students designing*. Software design is intimidating to most students, often leading to tentative and limited engagement. Any person, however, has an innate ability to generate ideas, solve problems, and refine solutions. We want students to immediately exercise these abilities, and become comfortable with them. This approach also sets the tone for the entire course as one of a hands-on and very open experience.
2. *To provide accessible examples about designing in which the students have a personal investment*. By taking pictures and movies of the students in action, these exercises allowed later examples to draw upon the students' own design sessions to make points about designing. This proved remarkably effective, especially when illustrating discrepancies between a student's practices in these exercises versus those in software.
3. *To jar students out of their assumptions about software design*. Previous classes expose students to some notations that may be used in design (e.g., UML, architecture), but talk little about the surrounding process. By using non-software design exercises, we immediately emphasize a broader perspective that focuses on the creative process of taking a problem, generating ideas, and critically evaluating alternatives.

The exercises and materials provided are carefully put together to illustrate some pointed lessons. The series of non-software design exercises typically moves from an individual exercise involving a purposely vague prompt (design brief), to providing a constrained problem, to a team exercise, to a team creating design instructions for another team to follow. Materials include pen-and-paper, Play-Doh, wooden sticks, and electrical wire, in different amounts depending on the exercise.

One of the exercises is to design a bridge with limited materials, a span as long as possible, and still able to carry a small load (a full can of Play-Doh). Another exercise is to design a classroom chair for mass-production. From these and other equally simple design exercises, a tremendous number of design lessons can be gleaned. Students naturally use different media (from prototyping a chair in Play-Doh to drawing a design on paper), they are creative (no two designs are the same), they design at different levels of abstraction (in addition to visual looks, some chair designs include measurements), they overcome communication barriers, they observe how different team members have different assumptions but also bring to bear individual knowledge that can be leveraged, etc. It is these lessons that we echo back when the students design software, and these lessons for which we provide them with the theory to put in perspective and the tools to effectively put in practice.

3.2 General Design Theory

After the introductory non-software design exercises, and before switching to software, the students are introduced to a theoretical, general model describing the key factors involved in design […]. The authors derived this model after studying design in a number of disciplines. The model illustrates key aspects of both the design product (e.g., it is an abstraction describing some eventual desired outcome; an outcome must be feasible; any design is but one in a large space of possible designs) and aspects of the design process (e.g., it involves ideas, goals, knowledge, and representation, all of which are continuously transformed into each other by various design activities; tools can only assist in generating and interpreting representations; representations are stated in one or more languages). The model is purposely not prescriptive. Instead, its descriptive nature enables different slices through the model to illuminate different processes, both conscious and subconscious, that typically take place in design. Jones' "divergence – transformation – convergence" is one such process [10], as is Schön's reflective conversation with materials [11]. Other examples abound.

Use of a theoretical model in class takes some of the mystery surrounding design away from students' minds. It illustrates that it is not black magic, but that there are clearly distinguishable factors and approaches that students can understand and practice. Moreover, by reintroducing video and photographic snapshots of their own non-software design exercises, we use the model to illustrate that they already have some naturally-occurring designerly behaviors. We complement the introduction of the model with assigned readings, including Spector and Gifford's paper relating software to bridge design [12] and book excerpts from seminal design authors such as Petroski [13], Jones [10], and Schön [11].

Use of the model also enables us to explain later in the class why we engage in various kinds of exercises, what kind of behavior is expected of the students in these exercises, and how they might go about the exercises. As such, the model really helps in removing students from existing biases and perceptions about software design (e.g., as a UML-centric rote translation of requirements into a certain code structure). In its place is a framework through which it is possible to precisely articulate designerly behavior and illustrate the challenges that the students will face as software designers, in this class, and later in their professional lives.

3.3 System Design

We then switch to the topic of system design, which we introduce as the activity of deciding what the functions of a software system should be and how it should interact with the world. It is rare that a complete set of requirements provides all of the answers in this regard, and many decisions must still be made. Indeed, a software designer is often tasked to design a system for a client under certain vague guidelines, and will end up working together with the client to effectively design the requirements. It is in this realm that our class aims to provide a design perspective to the students.

From our theoretical framework and the broader design literature, we identified six important traits exhibited by effective designers across fields, which we wished to impart on our students.

1. *Creative initial exploration of design decisions.*
2. *Selection of an appropriate central guiding design principle.*
3. *Creation and comparison of alternative partial solutions.*
4. *Flexibility in changing an ineffective design.*
5. *Selection of an appropriate medium in which to explore, record, and present design ideas.*
6. *Effective use of others' previously used approaches and solutions.*

Accordingly, the design exercises that we assigned to the students were quite open-ended, allowing for several possible approaches to the design process, as well as a number of different solutions. For example, students first received an in-class group design exercise about *Peak-Seeker*, an imagined program for sharing information about mountain climbing routes. Groups were to devise no less than three initial designs, list the concerns at play, and present these results in class. The goal was not to design the internals of the system, but rather to understand the users' needs, and figure out the goals that should guide the software's development.

After this initial exercise, we presented examples of existing architectures and designs, and also discussed at length the issue of primary concerns in a given design problem. Finding and addressing these primary concerns, often through a small handful of guiding design principles and decisions, is a skill that students have to acquire. Such concerns can reside in a system's structure, context, communication, user interface, persistence, algorithms, and other aspects. We highlight how each such aspect may require different strategies, knowledge, and languages to achieve and then express the envisioned solution.

Students then practiced system design over several weeks, through a variety of different exercises, including poster presentations with critiques in class, more formal presentations, silent brainstorming, requiring alternative design approaches, etc. Much of this practice takes place in an open setting, so that all students are continuously and actively involved in some design aspect at all times – whether it is generating, describing, or reviewing a design. Much was also continuously tied back to the six objectives listed above and to the broader framing in design theory and non-software design.

3.4 System Design Studio Projects

In the last six weeks of the course, students were tasked with two large-scale group projects involving a structured sequence of activities for exploring the design space, as well as deliverables and presentations that were critiqued. Feedback was provided by other groups and the teaching staff, in a variety of ways, including formal reviews, written critiques, and informal comments made upon hearing presentations. As such, each design project was a rapidly iterative exercise, in which both the product and the process could be critiqued.

The first system design project was to create an educational cooking game. Students had to consider how a software tool might be used to teach this subject, and had to determine at what basic level to model the cooking activities. This provides a potent example of our delineation between system and implementation design; there are difficult modeling decisions to be made about this system long before the program's internal structures are considered.

The second assignment involved four groups, each presented with a different, open-ended design problem (one problem involved the design of the software for a GPS watch to track children, another software for office managers to track office workers' activities). In each problem, numerous open decisions as to what the functionality of the software should be must be made in the context of critical issues such as safety, privacy, conflicting users' needs, unexpected consequences, and ease of use. Students, thus, were forced to creatively work within the boundaries set, consider the problem from many different angles, and often had to compromise to come to some sort of solution. Throughout, again, the focus is on design decisions at the level of how the system should function, tradeoffs that must be made, team work, and promoting designerly thinking. In this sense, the actual outcome – in the form of a document and associated final presentation – was less important than the process that each of the groups of students followed. We therefore asked each group to document this process and hand it in before every lecture, so we could monitor and help steer the projects closely.

4. IMPLEMENTATION DESIGN COURSE

The second course shifted focus to more traditional software design subjects, focusing on how to create a roadmap for the implementation of a software system as well as how to create this roadmap such that subsequent maintenance will be eased. We are prohibited by space from discussing all aspects of the class, so we highlight one particular aspect and briefly summarize others. The interested reader is invited to visit the course web site, where all of the materials for both courses are freely available. (http://www.ics.uci.edu/~andre/teaching.html).

4.1 Software Aesthetics

We opened the class with the slightly unorthodox subject of "software aesthetics"; in short, what is it that makes one design good, and another poor? An intuitive understanding of quality is important in the education of any designer, and software designers are no exception. These lessons began with a theoretical discussion of ways in which people have expressed a sense of "good software design" in the past, defining principles, enumerating objectives, and providing strategies for design. This involved traditional issues of coupling, cohesion, and "-ilities", but also more subtle points such as the formality and readability of diagrams.

The discussion was guided by a five-step exercise where students created designs for an electronic version of the board game Scrabble. Steps involved designing a first version, critiquing other students' designs according to our pre-defined, precise review sheet, revising one's own design based on the critiques received, implementing another student's design, and finally discussing scenarios of modifications to one's implementation. Throughout, an emphasis was placed on changeability: from the beginning, students had to consider possible changes that could be made to the game. Especially because students had to implement another student's design, this brought a sense of urgency and realism into the class.

This particular exercise provided a wealth of opportunities from a learning point of view. Students scrutinized other student's designs and had to confront being scrutinized by others in the same way. Issues of understandability, usefulness, and overall elegance of the designs took center stage, as it is always another human who must consume a design – a point underlying almost the entire exercise. Qualities that were anticipated of certain designs did not necessarily materialize in the implementation of those designs – even among the designs that were considered by the class to be the best. We were able to publicly (though anonymously) discuss both the designs and the critiques, the latter being critical to set a bar for what we consider acceptable design quality. These discussions covered all lecture time in the first weeks, promoting (as with our first course) a setting in which students design from the start, in an open setting, with continuity in the exercises.

4.2 Subsequent Course Topics

After design aesthetics, we touched upon a variety of topics key to real-world design. For each topic, we made sure to balance theory (as appropriate) with practice. We also continued the emphasis on design-minded thinking throughout, as such thinking should prevail in implementation design as much as it does in system design.

- *Design Recovery* – Students were asked to reverse engineer a design for Calico [14], a freehand sketching tool. Students worked in teams of four, and it proved challenging (unsurprisingly); even with tools that automatically generate a UML diagram of the code, understanding the principal design decisions underlying the structure is difficult. However, first having students go through the exercise and then jointly exploring the code and highlighting design decisions that we considered good or bad enabled them to learn about design from a real system, one that was built using an existing drawing API and employed an event-based architecture.
- *Design Patterns* – Our discussion of this subject was largely traditional, introducing a number of patterns to the class, discussing their philosophy and purpose, and providing examples of their use. However, we also leveraged the students' familiarity with the Calico code to ask them to modify their recovered designs to include several patterns of their choosing, and to justify their choices. This provided an opportunity for a hands-on attempt at employing patterns, and in design evolution, without the need to code an entire system.
- *Components* – We also presented students with lessons about finding and applying existing software components, as much of today's software relies on such external components. In particular, students were introduced to the question of how to set up an evaluation framework for choosing a component. Accordingly, different groups of students had to research the addition of three features to Calico (speech recognition, gesture recognition, and a new underlying graphical framework), to be delivered by reusing an existing component and fitting it into the existing Calico code. Groups were not only asked to present their candidates and final selections, but to discuss also the search process and selection criteria they used.
- *Large Scale Software Design* – By necessity, classes of this kind tend to focus on small-sized projects. But we wanted to dedicate at least one lecture to the issue of scale: discussing the ways that a project's size affects the lessons we had presented. This involved a series of subjects, including the ways that the size of a project affects planning for change, the role of documentation, and maintaining a unified design vision.

4.3 Final Project

One larger-scale project occupied the final two-and-a-half weeks of the class. This project involved two large teams, each of which was given the same task of designing and implementing a version of Pac-Man, with the added feature of allowing additional players to control Pac-Man's four ghostly adversaries – from other computers. This project presented a slightly more realistic scenario to the students; there were 14 members on each team, and the project was fairly large for the time allocated. Our primary motivation for this approach was, quite simply, to force students to divide up the implementation effort. While a four-person project might allow a single strong coder to "take over", that would be infeasible on this project, and in practice both teams divided into subteams to tackle different parts of the code. This required committing to an early solution, handling the overhead of communication between teams, and integrating a project's code; challenges that are best addressed through effective implementation design.

During the project, students exhibited many of the skills we taught them. Both projects used design patterns and one group was able to effective use the pre-existing UCIGame framework. While one group struggled with the networking element of the project, the other group was able to dedicate two members to researching that aspect of the problem, and effectively integrated their work. The project provided many challenges, but also a more realistic, integrated application of several of the class's lessons.

5. DISCUSSION

Our experience in teaching both courses twice in the span of two years has largely been positive. Student responses, collected both informally and through course evaluations, show that the students liked the vision for the courses, pedagogical approach taken, class materials, and overall feel of the courses. A telling indication was obtained unexpectedly this year: when asked about the most useful class in their entire four-year curriculum, a third of the graduating seniors mentioned one or both of the courses that we taught.

We consider a number of choices we made critical:

- The general design framing, particularly with respect to the first few weeks of hands-on non-software design exercises;
- Continuous design exercises throughout the courses to keep the students practicing and engaged;
- Assignments that built upon each other for several weeks at a time (Scrabble and the use of Calico being prime examples);
- Grounding the design exercises in design theory to motivate the practical experiences; and
- The use of larger final design projects to integrate the lessons from the previous weeks.

As a result, students gained confidence early on and truly fostered a design-based attitude towards software. Additionally, they could understand why the courses developed as they did, as well as their relevancy to anticipated future careers. The courses were demanding, yet students bought into the concept, delivered creative, high-quality designs, and participated outstandingly in design critiques and class discussions.

We realize that none of this guarantees that they actually learned something. In new courses, such as the two courses described, this is generally difficult to assess. From personal observations, especially in how students' design behaviors and the resulting designs evolved from early in the courses to the final projects, we believe they learned a great deal. We also note that in many ways the activities they undertook were more important than the eventual deliverables. One of the teams essentially failed its final project in the second course. This experience, however, was rich with examples that we used to drive home a variety of points, points that otherwise we would not have been able to illustrate so pointedly.

Despite broad buy-in, a few students found the first class too "soft", disdaining the non-software elements and not finding the material practical enough. On the other end of the spectrum, some students who were not as apt at writing code had difficulties in the second class. This is to be expected, given the nature of the Informatics students, some of which are more interested in a "programming" career and some of which more interested in a "non-programming" career. As such, we believe we actually found the right balance, but recognize that adoption of such courses in a major with a strong CS focus may find some resistance from students, especially in the case of the system design course.

Finally, some caveats about our approach. We found that at times the physical layout of the classroom was crucial to our success. It is important that students be able to flexibly move around, join in groups, discuss, participate, and work on problems; we wished at times that we had a space more appropriate to these activities. We also note that our approach hinges on a small class size. We were able to look at all submissions, both preliminary and final, thoroughly and quickly. We held interactive discussions and indeed allowed all of the students to participate in presentations and critiques. In a class of more than 30 students (ours had an enrollment of 28 max), such an approach may become untenable.

6. REFERENCES

[1] A. van der Hoek, D.G. Kay, and D.J. Richardson, " Informatics: A Focus on Computer Science in Context," in *SIGCSE Technical Symposium on Computer Science Education*, 2005, pp. 551-555.

[2] IEEE-CS and ACM, "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering," 2004. http://sites.computer.org/ccse/

[3] R. D. P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp "A Guide to the Software Engineering Body of Knowledge," vol. 15, pp. 35-44, 1999.

[4] S. Kuhn, "The Software Design Studio: An Exploration," *IEEE Software,* vol. 15, pp. 65-71, 1998.

[5] J. E. Tomayko, "Teaching Software Development in a Studio Environment," *SIGCSE Bulletin,* vol. 23, pp. 300-303, 1991.

[6] D. A. Schön, *Educating the Reflective Practitioner*: Jossey-Bass, 1990.

[7] A. Baker and A. v. d. Hoek, "Examining Software Design from a General Design Perspective" in *ISR Technical Report*: UCI-ISR-06-15 University of California, Irvine, 2006.

[8] N. Cross, *Designerly Ways of Knowing*: Springer, 2006.

[9] B. S. Adelson, E. , "The Role of Domain Experience in Software Design," *IEEE Transactions on Software Engineering,* vol. SE-11, pp. 1351-1360, 1985.

[10] J. C. Jones, *Design Methods*. New York: John Wiley and Sons, Inc, 1970.

[11] D. A. Schön, *The Reflective Practitioner*: Basic Books, 1982.

[12] A. Spector and D. Gifford, "A computer science perspective of bridge design," *Communications of the ACM,* vol. 29, pp. 267-283, 1986

[13] H. Petroski, *The Evolution of Useful Things*: Alfred A. Knopf, Inc., 1992.

[14] N. Mangano, "Calico Homepage," 2008. http://calico.bhnet.us/index.php?n=CalicoNav.About