Mixing Domains: Architecture Plus Software Engineering

Ömer Akın[1] and Ipek Özkaya[2]
[1, 2]Carnegie Mellon University, United States
[1] http://www.andrew.cmu.edu/user/oa04/home.html
[2] http://www.andrew.cmu.edu/user/iozka/home.htm

Abstract. *Software engineering is a multidisciplinary area of knowledge combining competence in computation with at least one other area of expertise, typically in the domain of the applications being created. A course offering for students of engineering, architecture and software engineering at Carnegie Mellon illustrates the challenges and opportunities of cross-domain instruction. These include ontology, problem taxonomies, and instruction strategies.*

Keywords. *Computing education in AEC; requirement engineering;*

Computing as a Universal Approach

Software is no longer the purview of computer science alone. Software Engineering is a professional concentration finding new homes in a variety of other fields including architecture, civil engineering, and building construction. The graduates of these allied fields, often referred to as AEC (Architecture-Engineering-Construction), are motivated by problems which can be addressed better with computation as well as with better computation. Software engineers, while experts at their own tasks, know precious little about any given domain of application they create, whether it is architecture or some other field; while architects not trained in the field of software engineering, are, at best, dilatants of software development.

Today computation in the AEC area is just as prevalent as, if not more so than, any number of other domains, despite inherent difficulties of digitizing an area with little standardization, complex processes, diverse agents and chaotic operational practices. This is the result of efforts that have begun decades earlier. As a measure of this, consider the publication entitled Pioneers of CAD in Architecture (Kemper, 1985), as sample of the CAD cross section in the USA, two decades ago. This book contains 62 entries by practitioners using the digital medium, twelve universities conducting research and other CAD applications, seven “expert” individuals, and finally, two technology providers. The practitioners are dominated by technically sophisticated architectural practices, engineering firms, management practices (notably health, real estate, equipment and other design sectors), and design-drafting service vendors. Technology vendors are clearly underrepresented as the industry giants, such as Autodesk, are non-existent. Academic institutions are not accurately represented either, but to a lesser degree than vendors.

Currently, there are virtually no firms that are immune to digital technology use. Vendors of architectural software are much more numerous than 20 years ago but still, market conditions allow

for only a dozen names to surface to the top (including AutoDesk, Bentley, AutoDesSys, Archicad, Magicad, Aris, Nemecheck Systems, and so on). Research publications and organizations that solicit these have proliferated (CAD Futures, ACADIA, CAADRIA, eCADe, ICCC, and so on). If one were to make an estimate of individuals working on software development in the field, this would easily run in the thousands. The series of questions we entertain here include: Who are these people? Are they architects, computer scientists, engineers, or graphic designers? How does their training prepare them for the software design tasks that they are undertaking? Can this preparation be done better? In this paper, we are going particularly to deal with the last question.

Situated Computing

Computing has always been a situated endeavor. Invariably software is created by applying the knowledge of computing to a domain of problem solving, which is external to it. Today, some of the most important applications of computation, such as design, information management, HCI (human computer interaction) and entertainment technologies, have little to do with the conventional analytical-quantitative problem solving domains from within which computation flourished. Increasingly, computer scientists as well as those in radically different domains have to understand each other's problems.

Those who take this task of crossbreeding seriously, inevitably, return to understanding computation through further education, particularly as it can be applied to the new problems they encounter. These students have to study specific issues of extra-computing disciplines. Furthermore, if they do not possess a strong analytical or computation background, their likelihood of success in a computation curriculum raises important challenges and opportunities. These include studying problem areas such as: requirement modeling, usability design, systems design, business planning, maintenance support, design implementation, and verification and validation testing.

How to use software development as a problem analysis approach can provide new educational opportunities to students engaged in training for specific extra-computing issues. A case in point is the AEC industry. Requirement specification, an approach we use in addressing the education of students in AEC computing, helps them better understand the problem they need to solve. This constituted the content of our pedagogic approach to the Software Requirement Engineering course we offer to AEC and Software Engineering students (Section 5). In preparing this course, we adopted several emerging approaches in the field to our course content and delivery strategies.

Emerging New Educational Approaches

The increasing use of computing in practice is instrumental in providing new opportunities in the teaching arena. In addition to learning how to use applications and new programming languages, students need to acquire and assimilate all together new techniques for problem solving, understand the fundamental concepts of software development, and strategies for customizing tools to better fit designer needs.

Ontological view of design

The task of the educator is to help students develop knowledge as skills for understanding, predicting and controlling those factors of a design problems that can help them interrelate three critical factors: structure, function and behavior. Structure is the physical form and materiality of a design. Function is the purpose to which it is placed. Behavior on the other hand is what actually happens when the structure is applied to the function.

In designing software the structures we create

are actual lines of code that process information reliably and repeatedly. This structure lies in the organization of commands issued to the hardware through machine language. Software writers, these days, use higher level languages (like FORTRAN, C++, JAVA) to sort the variables of a problem domain and the values that have to be assigned to them to achieve predefined objectives. They use structures like iteration, recursive associations, procedure calls, abstract variables, schemata, object oriented entities and the like to achieve their goals. These skills are normally taught within the confines of computer science courses like "Fundamental Structures of Programming (15-211)."

Functional aspects of programming are inseparable from the learning about structures. All structures serve a purpose. Iteration for instance takes you to any given value on a sequence of values generated by inserting an instance on a number series into an expression. The function of such a structure (iteration) may be finding the cumulative sum of transactions (value) on any given day of a week, month, year, or decade (number series). However, in courses like 15-211, the functions are given as the definition of a problem and students are asked to discover the "correct" structures to satisfy these functions. There is some flexibility here. Discovery of structures that expand the power and scope of these functions are usually recognized as "low lying fruits" that help student's understanding of the domain of functions. This type of learning can help students understand the taxonomy of relevant functionalities in their field and how to satisfy them. However, this exploration of functionalities is rarely driven by a purely functional agenda; on the contrary, its reach is determined by the agenda of computational structures.

The ingredient that focuses the software designer's gaze on software design functionality is the knowledge domain of software behavior. Behavior represents the activities that are supported and created by software structures in fulfilling the functions for which they are designed (Chandrasekaran and Milne, 1985). Designers start from functions around which to structure their designs. Users exhibit behaviors using these structures trying to fulfill the functions which drive them. More specifically, having the functionality to know the count of a set of events occurring over a period of time (a week, a month, or a year), could enable us to organize our budgeting, inventory keeping, shopping, menu designing, performance evaluating and so on.

Our course entitled Software Requirement Engineering focus on this behavioral paradigm in design. Our premise is that design begins with imagining the behaviors of the users of a design (software in this case, but is no different than many other design fields such as architecture). Once we define these behaviors clearly then we go on to defining the functionality of the application to be created. This functionality creates the basis on which a designed structure must be based and tested against. If we know what behavioral purposes we want to serve then we can precisely define if we need a mere count of frequencies over a period of time or if we also want to know who creates the event being counted or if the period increment (days versus hours) we use is appropriate.

Our most important means to deliver this body of knowledge in our software requirement engineering courses are Unified Modeling Language (UML) and Agile Modeling (Extreme Design).

UML as a method of software engineering

The Unified Modeling Language (UML) is defined as "a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems (http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/index.htm: May 2005). UML uses mostly graphical notations to express the design of object-oriented software projects. Braun, et al. (2001), state that "as the strategic value of software increases, the industry looks for techniques to automate the

production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale." The success of these endeavors depends largely on the success with which the expected behaviors of software users can be modeled while modeling the functionalities and the structures of software. UML provides utilities, largely centered on its use-case and test-case technologies that enable software engineers to do this.

Agile Modeling and Extreme Design

It is likely that you have seen at least a little bit of extreme sports on television: young kids on roller-boards or dirt bikes, making loop-de-loops in midair without the benefit of safety nets. Extreme design takes its name from the genre of things that are of that ilk. However, the intent of extreme design is far from those of its sports cousins. It is not for the thrill or even the challenge that is involved. It is to respond to needs that are immediate which demand the kind of agile thinking and acting that is required in most popular X-domains.

Extreme, or agile, design is not an entirely novel idea. Software engineers have been at it for a while (Ambler, 2002; Boehm, 2002; Constantine, 2001). The agile approach, also called Agile Modeling (AM), is a philosophy for software design that emphasizes new set of values, principles, and practices, rather than the prescriptive processes of the old (http://www.agilemanifesto.com/: May 2005):

We follow these principles (paraphrased) from AM:

- Welcome changing requirements, even late in development.
- Deliver working software frequently.. with a preference to the shorter timescale..
- Build projects around motivated individuals… and trust them to get the job done.
- The most effective method of conveying information is face-to-face conversation
- Simplicity -- the art of maximizing the amount of work not done -- is essential.
- Best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on ... and adjusts its behavior accordingly.

AM provides rapid and meaningful turnaround of design products by incorporating continuous feedback in the design process from the users and other experts. At a minimum, it provides an alternative to conventional and more cumbersome processes. The "Waterfall Model," during which every stage of work must be completed before the next stage can be started, dominates the conventional approached to software design. Such approaches have the distinct drawback of costly backtracking due to volatile requirements, particularly if the design process takes a long time to complete.

AM approaches do not run the same risks since they work with very rapid turnaround cycles. As more cycles of meaningful input from the client and consultants are incorporated, the quality of the final design is expected to improve (Beck and Boehm, 2003).

Problems of Computation in AEC

The past three decades of accomplishments in AEC fields demonstrate that there is an accumulated body of different computational problem types that reoccur in different AEC contexts. Research has brought to light an increasing number of examples about how to apply computational techniques to practice. In the traditional application areas of CAD we find physical modeling, databases, and generative strategies, prominently placed.

Physical modeling has been the central focus in the field ever since Sutherland (1963) developed his early application, Sketchpad. The illusive prob-

lem of how to develop universal digital media that would be able to capture all of the diverse representations of architecture continues to challenge and, to a substantial degree, elude CAD researchers and vendors. While there are important breakthroughs, particularly in the area of complex modeling and visualization of non-rectilinear building designs, this remains a challenge that invites novel solutions even today (Akın and Moustapha, 2004).

When it comes to automating graphic design, databases are the other side of the coin. Smart modeling systems rely on smart parsing of the objects being modeled. This varies from application to application, applicator to applicator, and so on. As a result, smart and versatile data organization schemata are indispensable. Databases designed to do this, and, perhaps more importantly, standards upon which such data representation is built, are critical in the development of the field.

The next set of critical problems, in the field, we call generative strategies. Architectural design is so complex that without the help of automated management of the very large and volatile set of design considerations, not to mention the tedium of dealing with excruciating detail and accuracy required in contract documents, architects would find themselves quite helpless. This is particularly so against the backdrop of increased complexity of buildings and building systems, and increasingly demanding client expectations. Generative systems, initially, tapped into the emerging area of Artificial Intelligence (AI). A series of generative strategies were spawned out of the tools of AI and fabric of Architecture: shape grammars (rule-based systems), knowledge-based (expert) systems, genetic algorithms, and the like.

Today we have even a larger set of problems in the domain of CAD in architecture. There is greater interest in the usability issues (human computer interface – HCI), organizational management of design, non-graphic design synthesis, and complex communication protocols and strategies. Each of these areas comes with a set of lower grain problems and solution strategies. Innovative HCI tools, for example, can enhance the design process by enabling digital prototyping and fabrication of designs. Design management area subsumes problems of data collection and information access at the job site. Non-graphic design approaches aim to provide supportive services such as data modeling, tracking and facilitating collaborative design activities. Communication strategies have a wide application area ranging from team work in the office to project management at the site.

The collection of problems defined by these diverse set of areas constitutes the overall problem set for computing in AEC. The richness of contexts represented by them underscores the challenges engineers in the AEC industries face both as practitioners and during their formative years as students of CAD. Our experiences in teaching a course for this constituency, described in the next section, underpin this paper.

The Software Requirement Engineering Course

We developed an interdisciplinary graduate course which introduces techniques dealing with these areas. In addressing the need to provide students with a better understanding of computing problems in AEC fields, we separated our pedagogic goals into two. One is to acquire techniques to gather information about the problem at hand as it applies to computing. And, two is to acquire techniques to analyze the information at hand to carry that information into a computing solution, be it a new application to be written, purchased, adopted, or for some cases dropped.

Software Requirement Engineering (48-759)

This course is intended to introduce students to the art and science of requirement modeling in software engineering. Requirement specification and modeling are critical strategies used during the early phases of design that are becoming in-

creasingly important for the successful delivery of computer products. Studies conducted in the 90s have shown overwhelming evidence that significant proportions of software errors and failures are linked to poor requirement specification. (Leffingwell and Widrig 2000) Furthermore, the cost of recovery from these failures is proportionally greater than the same in any other stage of software development. This course introduces approaches, methods and tools of "healthy" requirement development. It attempts to sharpen student's skills in this area through case studies and hands on projects. Object Oriented (OO) programming is the underlying paradigm assumed in this class.

The course, by and large, uses Leffingwell and Widrig (2000) and Wiegers (2003) for requirement management topics. In addition, it makes use of two other sources, among others, by Si Alhir (2002) and Alistair Cockburn (2001) to reinforce the coverage on UML and agile modeling, respectively. The course covers three main focus areas. It starts of by introducing the students to requirement elicitation techniques by spanning various methods such as use case modeling, prototyping, ethnographic analysis, requirements workshop and cognitive walkthrough. The techniques introduced in requirement elicitation also provide a cross over for AEC students familiar with design charettes. In order to help students utilize requirement information continuously we introduce modeling strategies in UML. We conclude with process models emphasizing extreme design and agile modeling. We cover quality measures and technical methods for requirement specification; agile requirement management methods; validation; traceability, and managing change.

We offered this course for five semesters since 2001 to an audience of interdisciplinary students from different AEC fields and software engineering. In the last offering the course was organized as two mini course modules of seven weeks each, instead of a full fourteen-week course.

The UML Approach

We approach user centered design methods for requirement elicitation and representation thorough the UML approach. This helps students analyze their domain problems with a software development focus while conceptually connecting the AEC student to their tradition of "facility programming." Problem understanding begins with effective software requirement elicitation techniques. We specifically focus on software development for computer aided architectural and engineering design. We build on the AEC students' already developed sense of requirement management to excel in requirement elicitation. We focus not only on developing a better understanding of how to build software, but also on developing strategies to identify, analyze and classify the computing problems. Students develop skills which allow them to bring strategic tools into practice, separating functional aspects from non-functional aspects, and identifying constraints that they need to be aware of. Through these skills, students develop critical thinking and are able to match the types of software problems that can be solved with specific computational techniques.

This new emphasis on understanding the requirements of the problem at hand helps investigate powerful and lasting computational solutions (Akın and Özkaya, 2002). Ultimately, all of this helps improve situated computing technologies and design solutions.

Extreme Design Approach

Extreme Design and Agile Modeling approaches, due to their reliance on participatory design strategies, are well suited for situated computing. In our pedagogic experience, the students produced a collection of significant requirement specifications that penetrated not only a functional understanding of the problem but also the behavioral realm. Remarkably, this was accomplished without work on weekends or multiple all-nighters during weekdays. The stress levels never reached

those often found in the normal project course. Even more remarkably, neither the intensity nor the quality of the work suffered.

This is neither a proclamation of a miracle nor an oversimplification. There are well recognized mechanisms that account for the rapid production of high quality work in the extreme context. First, even in the face of extremely short deadlines, architects, and designers in general, are known to produce remarkable work. There is plenty of anecdotal, professional experience to substantiate this. Frank Lloyd Wright's Fallingwater, for instance, comes to mind. "Eventually when he designed the first scheme for the house," Edgar Kauffman writes, "[it] was also really the last scheme for the house…" (Pfeiffer, 1986). During a panel discussion during the 50th Anniversary of the construction of Fallingwater, held at the Fallingwater, in June 24, 1986, Robert Moser and Edgar A. Tafel, reported that the design emerged out of Mr. Wright's hands during the period it took Mr. Edgar Kauffman, Jr. to drive to Taliesin West, which was purported to be a total of three days. Faculty will freely admit that some students do their best work during the final week of a semester, just before the final review. Finally, we all know the charette tradition of the Ecole des Beaux Arts that required the production of fully rendered plan parti in a single 24-hour period.

We believe that all of this is possible simply because it is indeed possible to design, and design well, agilely. First, in the extreme mode, one has to dismiss insecurities, doubts and the beginning jitters. There is no time for any of these. Second, one has to quickly identify a viable approach. There is not time to fuss about the "long shot" or the yet, "next" alternative. A single "good enough" design will do.

Good enough designs are relatively easy to come by (Akın, 1997); but, given sufficient time, designers keep searching for other solutions, regardless. The tight schedule curbs the designers' appetite for extended exploration. It helps carve out shortcuts towards reliable solutions. This is an invaluable skill mastered by the professional, if not for necessity then for a sense of parsimony and intellectual economy.

Finally, we built the extreme design problems on each other. Through this process of accumulating successive design features we realized benefits that exceeded independently designed ones. Ideas developed before were encouraged to be recycled.

The extreme design approach also has its perks for the faculty. There is no occasion for psychoanalysis or babysitting due to failure anxiety. The rapid pace is the great equalizer that eliminates over-achieving or over-obsessing, just as it eliminated procrastination and designers-block. Design products turn out to be suggestive and evocative, rather than over-worked and labored. This, inevitably, leads to discussions about the context of the design reaching to levels of participation unprecedented in the normal project course environment. Faculty remained engaged with every stage of the studio work. Every week literally meant either the issuance of a new phase of the problem or the evaluation of a completed one. This heightened the intensity of student faculty interaction and moved it onto a more informal platform.

Lessons learned

Our experience in the class has shown that the increasing use of computing in AEC practice has lead to the emergence of significant prototypical problems with an impact in the classroom setting. AEC students, when introduced to established software requirement elicitation techniques, such as use case modeling, ethnographic analysis, vision building and root cause analysis; not only develop a better understanding of how to build software, but they also develop strategies to identify, analyze and classify the AEC computing problems they have at hand. They develop skills, which allow them to bring strategic tools into practice through an understanding of software engineering, sepa-

rating functional aspects from non-functional aspects, and identifying constraints they need to be aware of. These skills allow students to develop critical thinking skills early on and help them understand the types of software problems in AEC that can be solved with specific computational techniques.

Our extreme design strategy presents an educational model that is: (1) efficient in its use of design time towards the production of designs, (2) intense in engaging students in the design task at hand, (3) effective in cumulatively building on previous design work and ideas, (4) powerful in the frequent and elaborate input from the instructors towards the students intellectual development, (5) instrumental in dispensing with "in the box" behavior, and perhaps most importantly, and (6) innovative in bridging pedagogy with design behaviors that are explicitly connected to practice. This approach is particularly attractive for teaching software engineering in domains of physical design (such as architecture) in which a culture of the "agile" or "extreme" already exists. In this approach, there are numerous low lying fruits for educators that deal with cross-disciplinary issues. For the AEC industry these include:

Collocating the requirement specification methods of several central fields

Training AEC students in analytical and software engineers in situated learning

Connecting disciplines by software engineering synthesis of AEC analysis Emphasizing behavior modeling as for the focus of successful situated computing

We believe AEC students need more exposure to understanding and using situated computational techniques to be well equipped to address the challenges of the profession. Our software requirement engineering course demonstrates an attempt in combining the students engineering and design expertise with software engineering methodologies.

References

Akın, Ö. and Özkaya, I.: 2002 "Models of Design Requirement" in H. Timmermans (ed) Sixth Design and Decision Support Systems in Architecture and Urban Planning - Part One Avegoor, pp.14-26

Akın, Ö. and Moustapha H.: 2004 "Formalizing Generation and Transformation in Design" in J. Gero (Ed) Design Computing and Cognition'04, pp.177-196

Akın, Ö.: 1997 "Paradigms and practice of computer assisted early design" in R. Junge (Ed) Proceedings: 7th International Conference on Computer Aided Architectural Design Futures, Kluwer, Dordrecht

Alhir S. S.: 2002 Guide to Applying the UML, Springer, New York.

Ambler, S.: 2002 "Agile Modeling and Feature-Driven Development" Software Development, pp. 5-7

Beck K. and B. Boehm: 2003 "Agility through Discipline: A Debate" IEEE Computer 36 (6) 44-46

Boehm, B. W.: 2002 "Get Ready for Agile Methods with Care" IEEE Computer 35 (1) 64-69

Braun, D., Sivils, J., Shapiro, A. and Versteegh J.: 2001 Unified Modeling Language (UML) Tutorial Kennesaw State University; website: pigseye. kennesaw.edu/~dbraun/ csis4650/A&D/UML_ tutorial/

Chandrasekaran, B. and Milne R.: 1985 "Reasoning about structure, behavior and function" ACM SIGART Bulletin Archive, ACM Press, New York, 1993, p. 4-55

Constantine, L.: 2001 "Methodological Agility" Software Development (June) 67-69

Cockburn A.: 2001 Agile Software Development: Software Through People, Addison-Wesley, Harlow

Kemper, A. M.: 1985 Pioneers of CAD in Architecture Hurland/Swenson Publishers, Pacifica, CA

Leffingwell D. and Widrig D.: 2000 Managing Software Requirements a Unified Approach, Addi-

son-Wesley, Harlow
Pfeiffer, B. B.: 1986 Letter to Clients: Frank Lloyd Wright The Press at California State University, Frezno, CA, p. 3
Sutherland I E.: 1963 “Sketchpad A man-machine graphical communication system” Proc AFIPS 1963 SJCC Spartan Books Baltimore, MD
Wiegers, Karl E.: 2003 Software Requirements, Second Edition, Microsoft Press