working under a well-defined process and product cycle to produce software for a known client or market and deliver it on a known schedule. This *closed-shop software development model* is increasingly at odds with actual practice. Some of the discrepancies between the closed-shop development model and the needs of modern software development include:

- System requirements emerge as the clients understand better both the technology and the opportunities in their own settings. This often requires software development to be carried out concurrently with business re-engineering.
- The pervasive integration of information technology with business operations and products requires software design to comply with market, regulatory, and policy requirements that typically are not evident in the project-specific requirements elicited from the client.
- Products of interest are often distributed and embedded hardware/software systems, not pure software. Classical software development methods focus tightly on functionality, performance, and technical quality requirements; they are not well suited to respond to constraints arising from the context of the application rather than its specific requirements.
- Mobile wireless applications that run on handheld devices with limited resources in dynamically changing environments impose new requirements for dynamic reconfiguration and interoperability. They also revitalize old needs for efficiency and usability.
- Software, especially system-level software, is now being developed by communities of cooperating volunteers. In open-source software, for example, code is published freely and interested users critique it and propose changes. Quality arises through an intense, highly parallel social process with rapid feedback rather than a carefully managed process.
- Software development often involves globally distributed teams. Accommodating geographic and organizational separation of developers within projects imposes substantial constraints on product architecture, processes, tools, and communication regimes.
- Applications are often created by harnessing coalitions of existing resources that are not under control of the software developer. The resources include calculation, communication, control, information, and services; they are often distributed, dynamic, autonomous, and independently managed. They may be modified or decommissioned without notice to users.
- Software development is increasingly disintermediated – software is adapted, tailored, composed, or created by its end users rather than by professional software developers. Placing enormous computational power in the hands of end users raises the stakes for making software-intensive products dependable, understandable, and usable. These end users need to understand software development in their own terms, not the terms familiar to professional programmers; they particularly need ways to decide how much faith to have in their creations. Current software products do not support these users very well.

These new aspects of software development often require an *open-shop development model* that is a major departure from the usual closed-shop model, and the uncertainties associated with external policy constraints and externally-managed resources require correspondingly more sophisticated analysis.

Most educational programs underplay the significance of these changes from software development of a decade ago. For example, developers trained to deliver well-defined products to specific clients or discrete products for the open market are ill-equipped to deal with the shifting needs of opportunistic web-based integration and the expanding involvement of end users.

In either the traditional or the emerging setting, the point of greatest leverage on overall software quality is early in design, before the usual software development methods can be applied. Early decisions often commit resources that will incur costs throughout the project. Boehm and Basili report that “finding and fixing a software

problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase" and also that the uncertainty in cost predictions is largest during early design [5,6]. Most software development methods, however, are largely linear processes that refine an initial design. These methods pay only passing attention to evaluating a variety of design alternatives early in development, and they place scant emphasis on understanding the market, business, economic, and policy context that limits the space of acceptable solutions.

The essential challenges are world-wide problems. Although we describe these challenges in terms of specific examples from the United States, the overall implications are global.

1.2 Resulting Forces on Software Engineering Education

To respond to these forces, we must prepare software engineers to construct and analyze systems that are heavily constrained by contextual considerations in addition to the usual technical requirements. These issues can affect the design in profound ways, such as by requiring or limiting essential functionality (e.g., audit trails) or pervading the implementation (e.g., security requirements) or limiting the architectural options (e.g., structuring databases for privacy). Contextual requirements are much easier to deal with as integral parts of the requirement than as add-ons to an existing design. However, to incorporate contextual requirements from the outset, developers must understand and respect these requirements as much as they do the requirements elicited directly from the client.

Currently, most software developers are educated principally in tools and methods for writing, analyzing, and managing software. For example, the ACM/IEEE Software Engineering 2004 curriculum design [23] devotes over 50% of its material to basic programming and analysis, about 25% to correctness and quality, 10-15% to process and management, and less than 10% to design. "Design" for this curriculum means implementing software to conform to requirements. The ACM/IEEE curriculum includes as guiding principles [23],

> *Reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations.* Students should engage in exercises that expose them to conflicting, and even changing, requirements. There should be a strong element of the real world present in such cases to ensure that the experience is realistic. Curriculum units should address these issues, with the aim of ensuring high quality requirements and a feasible software design.

and

> *Design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns.* Throughout their study, students need to be exposed to a variety of appropriate approaches to engineering design in the general sense, and to specific problem solving in various kinds of applications domains for software. They need to be able to understand the strengths and the weaknesses of the various options available and the implications of the selection of appropriate approaches for a given situation. Their proposed design solutions must be made within the context of ethical, social, legal, security, and economic concerns.

The curriculum itself, however, gives scant attention to these topics: about 3% for traditional requirements topics and 1-2% for contextual concerns in design.

1.3 Carnegie Mellon's Response

We believe that the greatest opportunity to improve software engineering education lies in improving students' ability to bridge the gap between traditional client-focused requirement elicitation and the capability-focused processes of software development – that is, in teaching them how to decide what to design. We created a new course to address these issues. Carnegie Mellon's software engineering tradition is strongly technical; compared to most software engineering programs, we place greater emphasis on engineering of the software product than on the development process. In addition. Our educational tradition emphasizes the enduring value of the content as well as the skills of immediate, possibly short-term, use. Details of both technical and nontechnical aspects of software engineering change with time, so we teach the enduring principles that support current practice as well as the current practice itself. This paper describes an innovative course in early design analysis rooted in the context of the Carnegie Mellon software engineering educational philosophy. We interleave sections from our statement of philosophy [38] with sections that describe our new course and show how it satisfies the principles.

Section 2 describes Carnegie Mellon's view on the content of software engineering. Section 3 presents our new course in the context of Section 2. Section 4 describes Carnegie Mellon's pedagogical philosophy, and Section 5 explains how the course satisfies that philosophy. Section 6 discusses experience with several course formats, and Section 7 suggests ways to adapt the course to other settings. Section 8 reflects on the role of this course in a modern software engineering curriculum.

2 The Carnegie Mellon Approach to Software Engineering

This section articulates Carnegie Mellon's core academic values for the discipline of software engineering. Curriculum design must reconcile academic values with the objectives of numerous other stakeholders; this is the case for the academic values stakeholder. This characterization of software engineering is informed by other software engineering and computer science curriculum designs, such as the ACM/IEEE guidelines [23], the IEEE SWEBOK [21], and the Carnegie Mellon Undergraduate Curriculum of 1985 [36], but it is independent of them.

2.1 Definition

Software engineering is the branch of computer science that creates practical, cost-effective solutions to computation and information processing problems, preferentially by applying scientific knowledge, developing[1] software systems in the service of mankind. Software engineering entails making decisions under constraints of limited time, knowledge, and resources. The distinctive character of software raises special issues about its engineering. These include:

[1] "Develop" -- Software engineering lacks a verb that covers all the activities associated with a software product, from conception through client negotiation, design, implementation, validation, operation, evolution, and other maintenance. Here, "develop" refers inclusively to all those activities. This is less than wholly satisfactory, but it isn't as bad as listing several verbs at every occurrence.

- Software is design-intensive; production costs are but a small component of product costs.
- Software is symbolic, abstract, and more constrained by intellectual complexity than by fundamental physical laws.

Software engineering is often confused with mere programming or with software management. Both comparisons are inappropriate, as the responsibilities of an engineer include the deliberate, collaborative creation and evolution of software-intensive systems that satisfy a wide range of technical, business, and regulatory requirements. Software engineering is not simply the implementation of application functionality, nor is it simply the ability to manage a project in an orderly, predictable fashion.

2.2 Core Principles

Software engineering rests on three principal intellectual foundations. The technical foundation is a body of *core computer science* concepts relating to data structures, algorithms, programming languages and their semantics, analysis, computability, computational models, etc.; this is the core content of the discipline. This technical knowledge is applied through a body of *engineering knowledge* related to architecture, the process of engineering, tradeoffs and costs, conventionalization and standards, quality and assurance, etc.; this provides the approach to design and problem solving that respects the pragmatic issues of the applications. These are complemented by the *social and economic context* of the engineering effort, which includes the process of creating and evolving artifacts, as well as issues related to policy, markets, usability, and socio-economic impacts; this provides a basis for shaping the engineered artifacts to be fit for their intended use.

These are the fundamental, pervasive, integrative principles that transcend specific details and characterize the field. They are core beliefs that shape our values about what things are important and how we as a faculty approach problems. These principles characterize the distinctive Carnegie Mellon approach to software engineering.

Physicists often approach problems (not just physical problems) by trying to identify masses and forces. Mathematicians often approach problems (even the same problems) by trying to identify functional elements and relations. Engineers often approach problems by trying to identify the linearly independent underlying components that can be composed to solve a problem. Programmers often view them operationally, looking for state, sequence, and processes. Here we try to capture the characteristic mindset of a software engineer.

2.2.1 Computer Science Fundamentals

The core body of systematic technical knowledge that supports software engineering is the algorithmic, representational, symbol-processing knowledge of computer science, together with specific knowledge about software and hardware systems. Major computer science principles include:

Abstraction enables the control of complexity. Abstraction allows selective control of detail and consequently separation of concerns and crisp focus on design decisions. It leads to models and simulations that are selective about the respects in which they are faithful to reality. It permits design and analysis in a problem-oriented frame rather than an implementation-oriented frame. Some levels of design abstraction,

characterized by common phenomena, notations, and concerns, occur repeatedly and independently of underlying technology.

Imposing structure on problems often makes them more tractable, and a number of common structures are available. Designing systems as related sets of independent components allows separation of independent concerns; hierarchy and other organizing principles help explain the relations among the components. In practice, independence is impractical, so issues of cohesion and coupling affect the results. Moreover, recognizing common problem and solution structures allows reuse of prior knowledge rather than reinvention. Software systems are sufficiently complex that they exhibit emergent properties that do not derive in obvious ways from the properties of the components.

Symbolic representations are necessary and sufficient for solving information-based problems. Control and data are represented symbolically, and this enables their duality. Notations for symbolic description of control and data enable the definition of software. These representations allow the description of algorithms and data structures, the bread and butter of software implementation.

Precise models support analysis and prediction. These models may be formal or empirical; formal and empirical models are subject to different standards of proof and provide different levels of assurance in their results. The results support software design by providing predictions of properties of a system early in the system design. Careful documentation and codification of informal knowledge provides immediate guidance for developers and a precursor for more precise, validated models.

Common problem structures lead to canonical solutions. Recognizing common problem and solution structures allows reuse of prior knowledge rather than reinvention.

2.2.2 Engineering Fundamentals

The systematic method and attention to pragmatic solutions that shapes software engineering practice is the practical, goal-directed method of engineering, together with specific knowledge about design and evaluation techniques. Major engineering principles include:

Engineering quality resides in engineering judgment. Tools, techniques, methods, models, and processes are means that support this end. They can enhance sound judgment, they can provide a basis for evaluating designs, and they can make activities more accurate and efficient, but they cannot replace sound judgment.

Quality of the software product depends on the engineer's faithfulness to the engineered artifact. This quality is achieved through commitment to understanding the client’s needs; it is evaluated by assessing the properties of the artifact that are important to the client. This is the basis for ethical practice.

Engineering requires reconciling conflicting constraints. These constraints arise both from requirements and from implementation considerations. They typically over-constrain the system, so the engineer must find reasonable compromises that reflect the client's priorities. Engineers generate and compare alternative designs and refine

the most promising; they prefer quantitative evaluations and predictions. Finding sufficiently good cost-effective solutions is usually preferable to optimization.

Engineering skills improve as a result of careful systematic reflection on experience. A normal part of any project should be critical evaluation of the work. Critical evaluation of prior and competing work is also important, especially as it informs current design decisions.

2.2.3 Social and Economic Fundamentals

The concern with usability and the business and political context that guides software engineering sensibilities is organizational and cognitive knowledge about human and social institutions. This is supported by specific knowledge about human-computer interaction techniques. Major socio-economic principles include:

Costs and time constraints matter, not just capability. The costs include costs of ownership as well as costs of creation. Time constraints include calendar (e.g., market window) as well as staffing constraints. These factors affect the system design as well as the project organization.

Technology improves exponentially, but human capability does not. Computing and information processing capability should be delivered to end users in a form that those users can understand and control. Systems should adapt to the users, not users to the systems, and the computing activities should fit well with users' other activities.

Successful software development depends on teamwork by creative people. Software developers must reconcile business objectives, client needs, and the factors that make creative people effective; they must communicate effectively with clients. Modern projects are too complex for individuals to handle alone.

Business and policy objectives constrain software design and development decisions as much as technical considerations do. Long-range objectives, competitive market position, and risk management affect the business case for a software development. Public policy and regulation add requirements that the client may not be aware of. These objectives should have equal standing with other objectives, such as technical and usability objectives, in the development process.

Software functionality is often so deeply embedded in institutional, social, and organizational arrangements that observational methods with roots in anthropology, sociology, psychology, and other disciplines are required. It is often relatively easy to capture the obvious functionality and constraints, but the subtle ones often go unnoticed and cause projects to fail or to incompletely satisfy users.

Customers and users usually don't know precisely what they want, and it is the developer's responsibility to facilitate the discovery of the requirements. Developers need to use appropriate techniques to help the customers and users explore the design space, and understand the relevant alternatives, constraints, and tradeoffs. This requires knowledge both of the technology and the context of use. Since most customers and users are unlikely to acquire substantial technical knowledge, developers must take the initiative to bridge the gap by working to acquire more than a superficial knowledge of the context of use.

2.3 Core Competencies

The fundamental material informs and pervades the curriculum. More visibly, the curriculum includes content, both mature and immature, that develops software engineering capability on the three foundations of core computer science, engineering, and the social and economic context. To describe the content, we develop a rough classification that allows us to plan curricula, to assess students' skills, and to identify intellectual gaps.

Software engineers should master a set of core competencies. These are abstract capabilities (e.g. "ability to reason in a formal system") not specific skills (e.g., any particular choice among CSP, Z, Larch, etc), and especially not skills in using particular products. It follows that different students may satisfy the capability requirements in different ways. So degree programs could be described with coverage requirements that refer to these capabilities. We might say, for example, that each masters student should demonstrate proficiency in reasoning with symbolic systems by using two such systems at some point during the masters program; this might be in a class, in a major project of the program, as part of an independent study project, etc. This model becomes increasingly important as the flexibility in the programs and the diversity of student activity increases. To support this, we envision a mapping from our educational offerings (courses, projects, etc) to these capabilities. Software engineers should:

- Be able to discover client needs and translate them to software and system requirements
- Reconcile conflicting objectives, finding acceptable compromises within limitations of cost, time, knowledge; understand the nature of unstructured, open-ended (sometimes known as "wicked") problems
- Design appropriate solutions, using responsible engineering approaches
- Evaluate designs and products
- Understand and apply theories and models that provide a basis for software design
- Work effectively in interdisciplinary contexts, in particular to bridge the gap between computing technology and the client's technology and to interpret and respect extra-technical constraints
- Work effectively within existing systems, both software artifacts and organizations
- Understand and apply current technical solution elements, including specific components, tools, frameworks, and also abstract elements such as algorithms and architectures
- Program effectively, including code creation, component use, and integration of multiple subsystems
- Apply design and development techniques as appropriate to realize solutions
- Organize and lead development teams, including team-building and negotiation
- Communicate effectively, both verbally and in writing
- Learn new models, techniques, and technologies as they emerge; integrate knowledge from multiple sources to solve problems; serve as a change agent for adopting new technology

For programs organized around courses, traceability from competencies to content could be performed for each course. For a self-paced project-based curriculum, the selection of specific topics may be driven by individual projects; in this case the traceability could be done for each student as a means for determining whether each student has satisfied the overall requirements of the program.

3 Course Content: Deciding What to Design

Section 1 identified the ability to bridge the chasm between client needs and the beginning of software development[2] as a gap in most software engineering curricula, in the capabilities of most software developers and, indeed, in the product design capabilities of most organizations that produce software. We developed a course to close this gap by teaching students how to handle the engineering design responsibilities that precede traditional software design – that is, we address the design tasks that move from requirements to set the stage for selecting a software architecture and applying a software development method. This section describes the course, "Methods: Deciding What to Design", and shows how it was shaped by the principles of Section 2.

Our course brings together a variety of methods for understanding the problem the client wants to solve, various factors that constrain the possible solutions, and approaches to deciding among alternatives. The course is principally intended for students in our professional masters' program in software engineering (MSE) [11]. The students are not expected to have previous knowledge in any of the topics covered in the course, but there is a prerequisite of having minimum three months hands-on software development experience in industry. The course lasts one full semester, 15 weeks, and is designed as a 12 unit course. This corresponds to approximately 12 hours of effort per week for students, including the time spent in class. The class size has been between 25-30 students. This program is built around a substantial ongoing project, the MSE Studio, in which students develop a software subsystem for a real client. The students take our course during their first semester on campus, at precisely the point when they receive the first, inevitably vague, statement of what the customers for the ongoing project want. As a result, we expect the students to find immediate uses for the course material.

As noted above, software development depends on bridging the gap between a vague statement of a problem and decisions about the specific components that make up a working software system. The challenge for software engineers is often bridging from the system-specific requirements and unspoken constraints from the operating environment to the high-level design of the system. This has been a persistent challenge for our students.

Most software engineering curricula have some courses that focus on requirements and other courses, for example about software architecture or object-oriented design that focus on the high-level software system design. These independent courses usually fail to address the nuances of recognizing tradeoffs, generating and comparing alternatives, identifying the implicit constraints that arise from the context of the project, and ensuring that the software product will be usable by its intended audience. Our course explicitly covers these contextual issues, and it requires students to apply their understanding of this material to the MSE Studio projects.

The overarching objectives of this course are for the students to be able to explain the major forces, both technical and contextual, that shape and constrain the solutions

[2] In our view, the ideas covered in this course correspond to the design phase of an engineering project. However, software engineering uses the word "design" to refer to the activities that begin with choosing implementation strategies such as software architecture.

to their clients' problems, to evaluate and address the ways these forces constrain the software implementation, and to select and apply appropriate techniques for resolving the constraints and selecting a preliminary design. Students should learn to handle easy cases themselves, and they should be prepared to interact constructively with domain experts such as business or usability experts for more difficult cases. Students should come to understand that good solutions come not from applying processes by the book but from genuinely understanding the client's real needs, then selecting and applying whatever techniques are appropriate to solving the client's problem.

In practice, the course is organized around five core competencies: identifying types of problems and their structures; eliciting technical needs; matching the design to user needs; understanding and analyzing business, economic, and policy constraints; and adopting an engineering approach to software systems. The course requires students to apply these competencies in the context of the MSE Studio project. We address each of these competencies in turn, describing the material in the most recent offering of the course. We have also taught the course with the order of Sections 3.1 and 3.2 reversed. We find, probably because of the coordination with the MSE studio project, that students respond better to the order presented here. Section 7 discusses alternative topics that could be substituted to make a comparable course suited for somewhat different audiences.

3.1 Identifying Types of Problems and Their Structures

In this section of the course, students learn to identify types of computing and information processing problems and their structures by studying a vocabulary of common problem types. We use Jackson's Problem Frames [22] approach as a way to identify the elements of a client's problem, identify pertinent properties of these elements, and recognize classes of similar problems whose well-established solutions can provide guidance for the problem at hand. Students learn to identify distinct concerns, or domains, of the problem and determine the characteristics of each domain – for example whether it is under the control of the software developer, whether it is physical or symbolic, whether it operates autonomously or only through external control. After analyzing the problem domains, students identify common problem templates (called frames), such as device control, information display, or data transformation, that appear in the client's problem. Finally they formulate the criterion for demonstrating that the software they're designing in fact solves the problem. The problem frames describe phenomena in the problem space, but they are common problems for which solution strategies, and hence software design alternatives, are well known.

Problem frame analysis of problems demonstrates several design principles:

- Good designers draw on a rich vocabulary of well-understood design fragments. This allows them to map new problems to systems of known problem types.
- Common problem structures appear regularly, even across application areas. Recognizing them allows the software designer to apply existing knowledge about solutions (and pitfalls), which is usually more effective than developing new solutions from scratch.
- Precise, often formal, analyses provide insight into problem characteristics, and analysis techniques should be selected opportunistically to match the problem.
- Problem analysis should enhance the designer's understanding of the client's needs and yield a plan for showing that the problem, and each subproblem, has been solved correctly.

- Idealized templates provide good guidance, but they must be adapted in each case to handle the details of the problem at hand.

Problem frames provide students with a way to sharpen their own understanding of the client, to impose structure that will lead to solutions, and to map the new problem into more familiar territory. This is a relatively short unit, and we do not use supplemental materials.

The project requires students to perform a problem frame analysis for their ongoing projects. This assignment comes early in the semester, so we emphasize understanding the major domains of the client's problem and the types of the main problem and a few principal subproblems.

3.2 Eliciting Technical Needs

In this section of the course, students learn ways to discover what the system should actually do in order to address the users' evolving needs. Use case modeling and contextual design methods address these concerns.

3.2.1 Use Case Modeling

Use case modeling [2] provides techniques for identifying an appropriate system boundary, understanding the interactions of external entities with the system, developing the capabilities the system should provide, and understanding the domain the software problem is situated in. This provides an effective starting point for matching users' expectations with what the system should actually do. A significant advantage of use case modeling is that it allows the students to formulate alternative usage interactions at a goal level rather than a mere functionality level. Our approach emphasizes use case modeling as an elicitation technique that can be used early in the software design to develop an understanding of the problem at hand.

Use case modeling allows us to demonstrate principles of:

- Software should be designed around the expected benefits to the users. Use case modeling accomplishes this by capturing the goals of users, not just the system functionality.
- The full complexity of the users' domain needs to be captured. Use case modeling requires identification of the various classes of roles and external entities that need to interact with the system.
- Identifying anticipated interactions with the system helps to identify implicit requirements. Specifying needs in general terms often glosses over complex and subtle user needs.

Supplemental readings help to broaden the students' understanding of the context of use. In particular, Carroll provides an alternative view to use case modeling by showing how the use of scenarios can transform information systems design. Scenario-based design uses concretization; scenarios are concrete stories about use [13].

Through individual exercises the students experiment with reverse engineering a product for a partial use case model and developing a domain understanding of the system they are designing. The project requires them, with their team, to develop an initial use case model and reflect on parts of the problem that they were not able to capture with techniques provided in use case modeling.

3.2.2 Contextual Design

Contextual design provides a particular example of a method for discovering subtle user requirements, translating them into an initial design, and iterating with prototypes

and customer feedback in order to validate and refine the design. The method sets out techniques for conducting interviews with potential users, generating models that describe how the work is actually done and the context of that work, and consolidating the work models and organizing functionality in a way that conforms with the ways users actually work [4].

Contextual design is firmly grounded in the tradition of social science methods for discovering social, institutional, and organizational phenomena. It prescribes a modeling discipline that resonates with software engineering students and helps to make the social science techniques accessible to them. Finally, it provides a coherent and comprehensive set of techniques that go from initial conception to a prototype solution. We present contextual design as an illustration of several general, enduring principles that guide early design:

- Representing the context of use requires appropriate techniques. Assumptions and prejudices about how users will interact with the system are an unsound foundation for design.
- Interactions with users in the design process should be carefully planned, and the interactions should center on concrete instances of the user's work. Simply talking to users, or just asking them what they want is unlikely to yield satisfactory results.
- Users are experts at their tasks, and system design must allow them to exploit their expertise.
- Good design usually results from informed exploration of alternatives, not from simply adopting the first solution that presents itself.
- Design of interactive software generally implies a redesign of how users work, with attendant risks to the user's ability to work effectively. These risks must be recognized and managed.

Supplemental readings bring a number of related points into the discussion. Suchman [44] discusses the difficulty of embedding a plan-based model of the user in the system as a means of guiding interactions in the context of copying machines. She emphasizes the need to help the user understand the state of the machine in order to interact with it. Moody [28] and Kidder [24] provide inside views of the often chaotic and turbulent design process in real organizations and of the push and pull of business and social forces. Christensen [14] introduces the notion of disruptive innovations. He makes the case that the designer's attention should not be limited to what the clients are asking for at the moment, but rather it should extend to understanding the business and technical contexts deeply enough to anticipate the client's longer-term needs and to place intelligent technology bets. This is complemented by von Hippel's analysis of the evolutionary nature of development [46].

The project requires students to conduct actual data collection by identifying at least two potential users of the system they are designing, planning and conducting contextual interviews, constructing work models, and specifying the organization of functionality from a user's perspective – the "user environment design" in contextual design terminology.

3.3 Matching the Design to User Needs

In this section of the course, students learn to evaluate solution alternatives from a user's point of view. The section begins with a continuation of contextual design, focusing here on constructing prototypes and interacting with users in order to

evaluate them. Norman [29] expands on the theme of usability, exposing the students to a classical discussion of user-centered design principles and a wide-ranging collection of examples.

The course uses current examples of design as material to which the principles can be applied. Recently, for example, the class focused on the BMW iDrive system that uses a control knob and menus to access over 700 of the automobile's non-critical functions. This interface provides a wealth of material for analysis of affordances, focus of attention, mappings, design for error, and mental models. This example also sets the stage for understanding the significance of business drivers – the initial version of the system was extraordinarily difficult to use and expensive to correct.

These materials provide students with concrete examples of techniques that embody several principles:

- Designing for users is an iterative process where interaction between user and designer focuses on meaningful, concrete representations of the designed system, presented to the user in context.
- Determining an appropriate level of prototype fidelity is a tradeoff between the cost of the prototype and the accuracy and precision of the feedback. For many purposes, low fidelity prototypes, such as paper and pencil mockups, are most appropriate.
- Software design requires simultaneously learning from codified experience and deeply understanding the unique subtleties of each particular design problem.
- The key to usability is creating a design in which the user correctly, continuously, and effortlessly maps the perceptual experience of the system to the user's mental model and the user's preferred way of working.

Supplemental material enriches the discussion with additional points of view. Brown and Duguid [8] offer an analysis of the difficulties inherent in separating knowledge from people and the inherently social nature of information, ideas that are critical for effective information system design. CSTB [15] offers a variety of ideas and research directions that focus on making computing available to a broader group of users. Winograd [48] shows how classical design knowledge applies to software, and Waldrop [47] provides a history of the creative thinking that transformed computers from remote mainframes used by specialists to ubiquitous tools in "human-computer symbiosis" with their users, connected intimately to daily work. Finally, Snyder [43] provides a comprehensive look at paper prototyping and its many applications in early design.

The project for this section requires students to construct at least two different prototypes for two focus areas of the user environment design that was the final product of a previous project. They then evaluate these alternatives, applying Norman's principles [29] and explanatory concepts in a process based on cognitive walkthroughs. If we had sufficient time available from project clients, we would prefer to have the students evaluate these prototypes with interviews conducted in the users' context.

3.4 Understanding and Analyzing Business, Economic, and Policy Constraints

In this section of the course students learn about the contextual forces that arise from the economic and business settings of software development projects. This section reviews elementary financial concepts and discusses the ways business considerations

can dominate factors in software design decisions. It covers ways to predict the value – benefit net of cost – of a software system by analyzing early design representations. The course exploits an example of current interest, such as privacy or internationalization, as a setting for understanding legal and policy constraints.

By treating software development as a value-creating activity, the course provides a framework to relate technical and contextual factors in software design decisions. On the one hand, students see how economic models such as utility theory can guide software design selection. On the other hand, they see how business, economic, and political requirements affect the kinds of solutions that will be acceptable. For example, international differences in privacy regulations should be anticipated at the point of database design, so that information that is regulated differently in different countries can be tagged or isolated. No textbook is available to cover all this material. We use Shapiro and Varian [35] to show students how economic concepts show up in the software market, but we must rely on a selection of papers for the rest of the material [19, 25, 27, 32, 33, 34, 39].

The material supports these principles:

- The objective of software design and development should be value to the company or client, not merely functionality. This requires analyzing both lifetime costs and benefits
- Analysis of value must take into consideration risk and time value of resources.
- Early, careful evaluation of designs makes software development more efficient.
- Many contextual requirements affect software design in fundamental ways; they should be addressed early in design.
- Decision models used in economics and social science can be useful for software design.

Supplemental material elaborates several aspects of this section. Shapiro and Varian [35] and Cusumano and Yoffie [18] show how market and competition shape strategic design questions. Two CSTB studies [16, 17] explore social and international issues in access to computing. Lessig [26] examines the tension between the original concept of the Internet as a commons for ideas and the use of the Internet as a commercial marketplace.

The project for this section requires students to identify one design issue for which the designer must consider business or economic constraints in order to get a good outcome for the client. They must identify two feasible solutions or approaches and compare the values of these alternatives, *as the client will see them*, applying topics of this unit as appropriate.

3.5 Adopting an Engineering Approach to Software Systems

In this section of the course students compare software engineering to traditional engineering disciplines to gain perspective on engineering practice, especially the need to reconcile conflicting constraints with limited time, knowledge, and resources. They study the nature of software systems, including the difference between program and product, issues of embedding, and the responsibilities of engineers. A discussion of the engineering approach to solving problems, both in software systems and in engineered systems more generally, complements this view.

By reading Brooks [9], students are reminded of recurring characteristics of software engineering projects such as manpower, scheduling, and second system

effect. They also see how other engineering disciplines draw on codified knowledge whenever possible.

The core principles we develop in this unit are already well recognized in other engineering disciplines. For example, design for scalability, situated design reuse, evaluation of designs in adapted contexts, attribute dependency and codifying design knowledge are common engineering techniques. The key principles we study are:

- Engineering entails making decisions with limited time, knowledge, and resources. However, problems often recur. Recognizing the recurring patterns and extracting knowledge from a codified knowledge base are typical of an engineering approach.
- Using a codified engineering base effectively is only possible when engineers understand the implications of changing scale and problem context on designs.
- Collection of relevant science and empirical results occur over time and experience. The use of these resources help engineers craft solutions to routine problems. Software engineering is now mature enough to start accumulating such resources. Recognizing the repeating routine problems in software engineering and driving the core knowledge from them is essential for the establishment of the software as an engineering discipline.
- Engineers must evaluate the complexity of their designs when they are put to use. They must recognize the dependencies between different components of their designs and evaluate their outcomes for human aspects.

We rely on supplemental material to provide concrete examples of engineering principles. We begin with material from computer science. Hoffman and Weiss [20] review Parnas' fundamental contributions to software engineering through topics as relational and tabular documentation, information hiding as the basis for modular program construction, abstract interfaces that provide services without revealing implementation, and program families for the efficient development of multiple software versions. Simon [41] explores design as a science; in particular he helps establish how software does admit to the same sorts of science as the natural world.

We complement this with reflections on older engineering disciplines, including aspects from civil, architectural, chemical, aeronautical, and mechanical engineering. Petroski [31] offers case studies showing the importance of understanding the context a design will operate in. He emphasizes the criticality of recognizing why and how errors occur in advancing the engineering methods for creating innovative solutions to problems. Vincenti [45] explains how engineering knowledge accumulates and describes how engineers use this knowledge in problem solving. Akin [1] shows how the architectural design process can be formalized and codified; he shows how expert civil architects go about solving their problems. Perrow [30] helps expose the social aspects of engineering design, especially in dealing with high-risk technologies. He presents complexity, especially tight coupling of subsystems, as a source of unpredictable cascading failures.

Students are expected to recognize the fundamental engineering principle discussed in the material and apply it to software engineering with examples either from their projects, past experience, or other units of the course.

4 Pedagogical Principles

The course we describe is a core course in a professional masters program. Many topics compete for attention in the curriculum, and many activities compete for

faculty and student time. The Carnegie Mellon software engineering faculty regards education as an investment from which students should reap benefits for decades. These pedagogical principles guide our curriculum and course implementations [38].

University education must provide knowledge of enduring value together with immediate competency. Universities walk a careful line between education in enduring principles and training in vocational skills. The Carnegie Mellon faculty believe both that a graduate should have certain competencies and that the investment in education should continue to pay off over a long period of time, and accordingly we ask our courses to serve both ends. There is ample evidence that the two are compatible, because students typically learn the principles best by working out examples that apply those principles.

In engineering, tools and skills cannot replace judgment. Engineering requires finding cost-effective solutions from among many and diverse alternatives. Methods, tools, processes, skills, heuristics, and other tools and techniques can help to organize the solution search to concentrate on good candidate solutions. These engineering tools and techniques remind you to consider possibilities that you might otherwise ignore. They can help a good (or even adequate) engineer find better solutions more effectively. They are not – and cannot be – a substitute for actually understanding the problem and making sound judgments about solutions. An intrinsic characteristic of engineering is the requirement to strike appropriate balances among conflicting goals. So pursuing one tool or technique to the exclusion of all others is only very rarely, if ever, appropriate. Above all, we should teach our students engineering judgment and the commitment to use it; all the specifics support this end. In particular, as students in our class apply their newly-acquired knowledge to the studio projects, faculty and other students facilitate reflection by providing constructive critique of their efforts.

The Carnegie Plan provides excellent guidance about engineering education. Each student should learn not only specific content, but also the principles and mindset of the profession, the ability to learn new material independently, and the perspective and judgment to be a responsible adult. In particular, graduates should be able to assume responsibility for their own continued professional development. Therefore, they should learn not only today's methods and technologies, but also the underlying principles and critical abilities that will allow them to select and master new methods and technologies as they emerge. This idea is captured in the Carnegie Plan, which Carnegie Mellon established half a century ago as part of a major restructuring of engineering education, both at Carnegie Mellon and throughout North America (see Appendix). Carnegie Plan carries institutional memory for our university. It is also an enduring statement that provides guidance for blending theoretical understanding and practically focused experience into durable skills and the ability to learn new material.

Hands-on, attentive time on task is critical to learning. Simon showed us that the strongest correlation between demonstrable learning and any of the student and instructor activities is with the student's engaged, attentive time practicing with whatever was being learned/taught: “Learning has to occur in the students. You can do anything you like in the classroom or elsewhere – you can stand on your head – and it doesn’t make a whit of difference unless it causes a change in behavior of your students” [42]. He also told us that experts have indexed memory of 50,000 to

100,000 chunks in the area of expertise, taking 10 years to acquire this expertise. So a reasonable aspiration for a university course is to get a good start on the 50-100,000 chunks by providing the student with a conceptual roadmap, lots of hands-on practice in various parts of the domain, and the ability to fill in more of the chunks.

Curriculum design is at heart a resource allocation problem. The scarce resource is student attention, measured however imperfectly by courses, hours spent, pages of reading, numbers of projects. To provide the greatest value, we must require each course to contribute to both enduring value and immediate competency. This favors content backed by good theory, because good theories compress lots of content into tidy packages; however, this leverage should not be allowed to drive out important (but partially codified) content in favor of pure theory. On the other hand, extensive exercises in a process that is likely to be obsolete in a couple of years can (usually) only be justified to the extent that they support long-term knowledge. It is easy to identify "important" content that more than fills the space in the program -- whether space is measured as class time, student attention, hours of work, or something else. It is much harder to set the priorities that lead to a curriculum that strikes the right balance of coverage and depth.

Sampling is sufficient; it is not necessary to cover everything. For students of the high quality that we admit, thorough mastery of a few exemplars coupled with principled overviews should provide a sufficient basis for learning other related material. We must take care, though, to provide sufficient coverage. This is a reasonable choice because curriculum space is a scarce resource and, per the Carnegie Plan, our students can assume responsibility for their own professional development.

Admissions should be selective, and we should make every effort to help admitted students succeed. The overall quality of students in a class affects the level at which the class functions, especially in interactive settings. When, as in many of our activities, students work in teams, the quality of each student's experience depends on the quality of the other students. Further, failure of some students affects the whole community of students. It follows that we should attempt to admit students with a good chance of success and commit to making them successful.

The educational setting should enable students to learn effectively. Learning depends chiefly on the active engagement of students, who make a substantial investment of time and resources. We should provide opportunities and resources that allow students to do this effectively. While providing sufficient resources, we should at the same time provide a realistic development setting

5 Course Organization: Engaging Students Actively

Our "Methods: Deciding What to Design" course is a core course in the MSE program at Carnegie Mellon. This is a terminal degree program for students with several years of software development experience; its objective is to enable its graduates to practice software engineering at a very high level of proficiency and to serve as technical leaders and agents of change in their companies. In implementing this course, we considered the type of students who enter our professional masters

programs and the principles of Section 4. We also considered other courses in our curriculum and ways the full suite of courses plus the Studio complement each other.

This course emphasizes mastering the material for use in practice. It won't serve the MSE program or the broader pedagogical principles if it is taught as a set of facts and skills. It can only work if it engages students in thinking hard about real – not textbook – problems. To that end, we teach the course by engaging students with real problems in as close to a real-world setting as we can arrange.

In terms of the Bloom taxonomy[3], we are principally interested in mastery at the higher (analysis, synthesis, and evaluation) levels. It follows that the objectives for our courses include a combination of "understands" verbs and "can-do" verbs. Further, the "doing" parts of the course must support the "understanding" parts. We probably all agree that courses in which students can hack their way to apparent success aren't serving their ends. It also follows that reflection and interpretation are more important than extensive routine drilling, comprehension more important than highly technology-specific skills.

The course takes advantage of a key feature of the MSE program – the MSE Studio. The entire program emphasizes application of course material in a practical hands-on experience, and the MSE Studio is built around projects for real, paying clients who expect actual deliverables. These year-long Studio projects provide students with a real software development setting where they must overcome technical challenges, understand new problem domains, manage team dynamics and client interactions, formulate problems, and – in the end – deliver software products to their clients. The course described here relies on the MSE Studio for real (not simply realistic) examples, and observations about how they apply course material to the Studio provides valuable course feedback. The use of real projects allows students to develop immediate competency to critique and apply the techniques they have learned rather than only acquiring textbook knowledge. Section 7 discusses ways to teach a course such as this in the absence of a Studio.

5.1 Homework

We emphasize applying and interpreting the readings, not simply doing small-scale exercises. Homework assignments for each of the readings help students focus on the aspects of the reading that are important to the course. Moreover, students do the homework based on the reading material and the class is used for discussion and interpretation, in contrast to the traditional format of presenting material in class as a basis for the homework assignment. Some of the homework questions ask students to answer questions on the day's reading; others ask them to apply class material to a problem or to their Studio project. The recurring objective verbs in our assignments are "understanding", "applying", "assessing", "discriminating", and "identifying". By

[3] The attributes of the Bloom taxonomy [7] are:
- Knowledge: remembering previously learned material.
- Comprehension–: understanding the meaning of material.
- Application: using learned material in new and concrete situations.
- Analysis: breaking down material into component parts to understand its structure.
- Synthesis: putting parts together to form new wholes.
- Evaluation: judging the value of material.

clearly explaining the objectives of the assignments, we both focus the students' efforts in answering the assignments and remind them of the enduring values to take away from the course [40]. We use sampling as a technique to balance coverage with providing enough hands-on experience for understanding the topics. By limiting their answers to a page or two in length, we try to help them focus on the most important aspects of the topics. Students find this more challenging, but we see them engaging the material at higher levels of the Bloom taxonomy, especially in class discussions. When an assignment asks for answers based on the Studio project, students work with their project groups on that assignment. However, they must answer the daily homework assignments individually to show their own understanding of both the project and the class material. The homework assignments provide individual hands-on attentive time on tasks for students. It also provides the team with multiple points of view of the same approach, allowing them to evaluate alternatives against each other when they need to apply the same technique to their project as a team.

5.2 Real-Life Projects

In each unit of the course, the project groups apply ideas of the unit to their MSE Studio projects and report the result to the class. Reporting to the class includes making a short 8-minute presentation of the main points, leading class discussion, and providing a 2-3-page summary of the major ideas. Students start working with the clients for their Studio projects the first week of the class. The projects help the students develop preliminary results with their clients; they can subsequently build on these results as part of the Studio. This allows the students to rapidly evaluate the effectiveness of the methods they learn in class for their project. We expect them to reflect on their experience, which also provides them an opportunity to recognize the core competencies set out in Section 2.

Each unit runs between two and three weeks. Through project assignments we address curriculum resource allocation and sampling, in addition to providing further opportunities for hands-on practice. Students get an opportunity to immediately apply what they learned to a selected part of their project. This first experience does not make students experts in the area of the assignment, but this exposure provides a basic understanding they can build on later, in accordance with the Carnegie Plan.

5.3 Communication Skills

The MSE program places a strong emphasis on communication skills. In designing the Methods course to include significant numbers of student presentations in class, we are both relying on the communication skills the students have already developed and providing more opportunities to exercise and further develop their skills. Each student presentation uses almost 1% of the class time in the course, and we depend on each student to contribute comparable value through his or her presentations. Since the purpose of each presentation is to explain to the class what the main idea of the topic is and how that idea should affect software design, we expect clear presentations that make the connection between the background material and the class projects. These criteria remind students not only to master the material presented in class, but also bring in their own experience and critical insight to the class.

5.4 Independent Interpretation

Most of the supplemental materials are books. We incorporate them in the course by assigning each book to a project group. The group is responsible for reading the book and reporting on how ideas from the book are related to the course. The reports on the books do not attempt to cover the entire book, but rather to identify and explain the most important points relevant to the course and software engineering in general. Similar to the projects, each report consists of an 8-minute presentation followed by discussion and a short written report from the group, 2-3 pages in length.

These independent interpretations provide another opportunity to combine the pedagogical principles of sampling and curriculum resource allocation. The supplemental readings and reports not only enrich the units, they also provide further sampling of techniques to which students can return when applicable. This activity also highlights the value of engineering judgment because we emphasize critical evaluation of the key take away of the books and application of the ideas to their Studio projects rather than a summary of the topic.

To help students prepare for the presentation and to focus the report, we ask them to provide a short abstract (approximately 50-75 words) of the viewpoint in advance. The purpose of the statement is not simply to identify the topic the author writes about -- it requires the students to identify what the author "says" about the topic. For example the statement should not be of the form "*the authors talk about the business of software...*" but "*the authors refute arguments that we need new economic models for software by arguing that the old models work, just with radically different parameters. They support this with discussions of adoption curves, lockin, ... For our studio project this implies, for example, ...*" We describe this as a statement that will help people remember what the book might offer them, so that they'll remember enough to find the book when they discover sometime later that they need it.

We expect critical analysis and independent interpretation of the selected books. Sometimes students have to try several times before they are able to explain how the reading applies to the course instead of just giving a book report. In the process they develop their engineering judgment skills as they have to think about how to report their conclusions to the rest of the class most effectively.

This exercise helps students learn to recognize outside material relevant to the course. It not only introduces them with a broader set of concepts, but it also gives them experience with evaluating techniques on their own. In addition, it allows them to recognize how to use different techniques together for the principles introduced in the course. For example for their projects students have in the past opted to use paper prototyping [43] in conjunction with either use case modeling [2] or scenario-based design [44] and evaluated their results through Norman's design principles [29]. The students are able to arrive at such rich combinations that cross over multiple units as a result of the variety of activities they engage in during the course.

5.5 Critical Evaluation

We expect students not only to learn the material they study, but also to apply it spontaneously to practical problems. In this way we require the students to evaluate which techniques work well, which techniques may work with improvement, and

which techniques may not be suitable for the task at hand. Reflective practice is entirely consistent with the Carnegie Plan's combination of education grounded in enduring principles, skills in applying these principles and continual improvement, and experience grounded in the real world.

The opportunities we provide for students to develop critical evaluation skills extend to all the activities of the class. Our evaluation criteria emphasize that we value insights combined with a student's experience over only reporting back what the lectures or readings already covered. In project reports we expect teams to evaluate how they have applied the methods to the specific project domain they are working in. We conduct the class meetings by providing ample opportunities for students to not only ask questions, but also contribute with their experience obtained either from other classes or from industry experience (simply repeating material from the reading without adding insight does not count as valuable discussion).

6 Experience and Evaluation

Our evaluation of this course is based on our own reflection, on the progress we and other faculty observe in the Studio projects after the students finish the course, and on the feedback we receive from the students (both while they are still in the program and after they graduate).

6.1 Our Evaluation

We have offered this course in lecture format four times. The third time we concurrently prepared the course for distance delivery, and the second distance offering is underway as this paper is under preparation [10]. After each offering we have reflected on student performance and participation and revised the course accordingly. Our own reaction is that the course addresses the concerns we had when we designed the course. This provides an element of face validity.

The faculty mentors of the MSE Studio report that since we have started connecting class assignments directly to the Studio projects, the students' performance in applying course material to the Studio has substantially increased. Changing the order to present problem frames first has enhanced this effect.

When real projects are introduced to the curriculum, the closed-shop software development model no longer holds even in the educational environment. The concerns we identified in Section 1.1 are challenges that software engineers need to learn to deal with in school as well. Our students have to live through the challenges of evolving system requirements as the clients understand both the technologies and the opportunities available to them. The contextual design, use case modeling, making the result useful, and problem frame analysis methods we introduce in the course provide opportunities to help the teams to help their clients understand their needs. This year all three projects that are assigned to our on-campus teams have requirements where the end product needs to be adapted or tailored even after it is delivered. Open source development and globally distributed teams is no longer a myth, but a reality. The business and economic considerations and problem frames analysis topic assist in surfacing challenges, consideration the teams must pay attention to.

6.2 Responses from MSE Program

The responses from the MSE program focus on improvements in how students get engaged with solving the problems for their Studio projects early on using the techniques introduced in the Methods course. Several teams opt to continue using the methods that they are only introduced to in the course in their Studio projects. The teams continue using these methods as a driver for their software development efforts in a longer term and larger context. They report that the techniques help them bridge the gaps between the client, the team and the technical challenges.

- A team of four students from the 2003-2004 academic year with members ranging from 2 to 10 years of experience, report that using paper prototyping, a technique they learned from independent interpretation of supplemental material, helped them scope a problem context that would otherwise have been hard to manage. This allowed them develop a technically competent solution in a short amount of time. Their success led other teams to try out the technique in subsequent years.
- All the teams each year employ advanced use case modeling as a problem understanding and requirement specification technique. The distance education students, who tend to have more industry experience than the on campus students, report advanced use case modeling as a strong technique to employ directly in industry, especially when coupled with contextual design – provided that economic resources are available.
- Two out of the three on-campus teams of 2004-2005 used problem frame analysis successfully to understand and refine the scope of their problem. Teams used this technique internally rather than sharing it with their clients, so they report their application as "informal". In the semester following this course, one team that was been struggling with representing the intrinsic details of their system with use case modeling revisited their problem frame analysis. They used problem frames together with use cases to identify different subsystems and discover which subsystems acting on other subsystems. They report that using problem frames to identify subproblems and their relations with each other helped them understand why their use cases were being criticized as trivial.
- One on-campus team of 2004-2005, after studying industrialization and versioning as part of the business and economic considerations unit, decided to address this and to their surprise discovered that this was one of their client's concerns, though it was not mentioned in client discussions before the students brought it up.
- One 2005-2006 team used cost/benefit analysis comparing an open source tool to a $60,000 commercial one They showed better long term return with the commercial tool. The customer bought the commercial tool because of this analysis. Discussions with the team members revealed that they were resigned to using the far less capable open source one, until they did the analysis as part of their class work.

6.3 Student Responses

Students with little industrial experience find it challenging to be asked to apply the material to an actual project, with all the ambiguity and hidden constraints that are typical of real projects. As they get more experience with the material, either by applying it to the Studio project or applying it on the job after they graduate, they find it very valuable. Students in the distance offerings also report that the course is both challenging and worthwhile. Distance students are usually working at the time they are taking the course, and they often report applying the material in their current projects. The feedback we get from students relate to several areas of the course organization.

Collaborating with other professionals. The revelation that a professional has to be able to collaborate with people with complementary skills comes as a surprise to students with little industry experience.

- Two students from Fall 2004 report that dealing with a team member who was more experienced than themselves in the problem domain was challenging. This even influenced how they perceived the applicability of the methods introduced; however, it also gave them an understanding of how they should have relied on using the methods in dealing with the experience gap.

Critical thinking. The course format comes as the biggest surprise to some students. Going from memorize-and-recite-back courses to critical application of content, learning to make public presentations, and learning to contribute to discussions is challenging at first. However, students start realizing the benefits as they are encouraged to conduct these activities regularly. We expect not only substantive results, but also communication and organization skills. We provide immediate feedback so that students can work on their weak skills before they have to deal with similar tasks again. Admittedly, this course is fast paced and does at points stretch the students if they come with insufficient background. On the other hand, students start seeing the benefits as early as a few weeks into the course. The benefits of frequent individual homework assignments with specific hands-on tasks -- at times frustrating due to time constraints -- are appreciated.

- One student from Fall 2004 reports, “In my undergrad, we had only final semester exams and we were graded based on that exams' performance. We would never get our answers back and hence never understood what mistakes we committed! The way it is done here is very beneficial and I really appreciate the way you put comments on each and every one's answers. This has been a new experience for me, and I am benefiting a lot.”
- One student from Spring 2003 reports “Honestly, after my first presentation, at first I did not even understand why I did not do so well. Your feedback on my presentation was really valuable for me, not only for grade improvement, but also for my communication skills.”

Formal exposure to known techniques. Students with longer industry experience are encouraged by seeing what they had to deal with in industry presented in a structured manner, and they also appreciate new techniques for dealing with familiar problems.

- A student from Fall 2004 with 5 years of experience reports that although he had to design and prototype many software products he never was introduced to approaching them with principles covered for making the result useful unit. He emphasizes that this exposure helped him evaluate how it can be done more effectively.
- A student from Fall 2003 with 10 years of software architecture development experience reports that he was very happy to have been introduced to Brook’s [9] principles after he graduated because there have been several incidents where he had to be reminded of the relationship between adding complexity and scheduling.
- A student from Spring 2005 offering who is taking the distance education version of the course reports he will take back contextual design to the rest of his company to use as a part of their software development process because it will help them identify product visions that are more in line with user needs. He also adds that they will first need to look very carefully into cost-benefit analysis of resource allocation.
- A student from the Spring 2006 distance offering reported, “Despite all of the moaning about problem frames, I think I ‘like’ it most of the three approaches so far. (Does this make me ill in the head?) It is more applicable to my work than Use Cases and Contextual Design. At [my company], I work with embedded systems that by nature interact with other

systems. There is no human actor, no user centric context. While use cases can have system actors, that isn't really strong suit. Contextual design is almost entirely focused on the user environment which doesn't exist in my systems. So consider this a vote in favor of problem frames. My only gripes really are that I believe it can be boiled down to a science and Jackson is almost there. He has 90% of the concepts down at 70% clarity."

- The distance education students of Spring 2005 and 2006 offering also had to deal with challenges of not being collocated. These students were current industry professionals who were attending the program while they continue to work. They embraced the challenges of the distance environment as a learning opportunity and report that this allowed them to evaluate how they can apply the techniques in an outsourced and distributed environment. The distance education students readily embraced problem frames, which students with less practical experience find one of the most challenging techniques. They report this technique could be very useful to bridge communication gaps when parts of a project are outsourced.
- Many students with extensive experience come into the course having applied, or at least having heard of use case modeling. A student from Spring 2005 distance education offering reports going through the process in a different way than the one employed in his company allowed him to improve his use of the technique.

7 Adapting the Course for Other Settings

This course is organized around the particular educational setting at Carnegie Mellon. In this section, we discuss our assumptions about the setting and about ways to adapt the course when these assumptions do not hold.

7.1 Selection of Topics

One major practical advantage of the sampling principle is that the contents of various sections of the course can be varied substantially while achieving the same educational objectives. There may often be good reason to choose example techniques to illustrate the important principles other than those mentioned here. Techniques may be changed because of their immediate applicability to a project or because a change complements the content of other courses.

We have taught this course four times as a lecture course and twice as a distance course. We have varied the specific techniques several times. For example, in the Eliciting Technical Needs section, we have taught task analysis rather than contextual design; in the Business, Economics, and Policy section, we have covered privacy and security in place of internationalization, and we have sometimes covered QFD [3]; in the Making the Results Actually Useful section, we have covered cognitive walkthroughs rather than paper prototyping. This kind of flexibility has allowed us to experiment with new content that we think will build immediate competency and to take advantage of available expertise in our environment.

7.2 Student Presentations

Our time budget for class meetings is heavily influenced by the need to schedule two presentations by each student. Typically, each student will make formal presentations of one independent interpretation and one project report during the semester. As a team member, each student will work on about five written project reports and five

written independent interpretation reports. As mentioned above, we consider the presentation part of the course as a key skill-building element as well as a way of bringing diverse content into the class discussions.

This approach does not scale well, of course. We typically have 25 or so students in the class, and adding significantly to this number would require cutting back on student presentations in order to allow sufficient time for other course content. One solution would be to have recitation sections for student presentations and discussion. Large classes could be divided into a sufficient number of recitation sections to introduce concurrency into the student presentation component. This depends, of course, on the availability of sufficient faculty and teaching assistant resources.

Similarly, very small classes with students fewer than 12-15 students would pose challenges as well. Small classes could lose from the content, especially with reduction of the independent interpretations from students.

A more challenging situation is presented by a distance education version of the course. Several approaches are possible here, given the availability of appropriate technology. In situations where students meet regularly with a distance education instructor, presentations can proceed more or less as usual. If there are no, or very few, face-to-face meetings, presentations could be done using some form of conferencing technology. To the extent that students have reasonable equipment available, this could be an effective option. The most critical pieces are good audio quality and a way of sharing presentation slides. For the Spring 2005 and 2006 distance education offerings of the course we used a web-based collaboration technology which allowed us to speak using voice over IP and share presentations via desktop sharing. With a class size of 10-13 this technology worked smoothly; however, for larger classes it would pose challenges in providing students enough opportunities to participate and present in this kind of an environment. Another option would be to digitally record a student's presentation using some local resource, and providing the audio, slides, and perhaps video to other students online. Discussion could proceed by means of a discussion board (if students observe at different times) or chat session (if students observe simultaneously) in which the students (including the presenter) and instructor participate.

7.3 Student Prior Experience

Our students typically have a year or more of industry experience, and we have found that this experience is critical for success in the course. Less experienced students often struggle because they don't have the same level of intuitive understanding of the problems of large projects with real customers. This need for experience has been reinforced in observing the distance offerings of the course. The distance students have, by comparison, an average of over 5 years experience and did not have any of the problems associated with some of the minimally experienced students on campus. Additionally the discussions, albeit asynchronous through a discussion board tended to be more robust and insightful. While part of this is undoubtedly due to asynchronous discussions providing more time to ponder questions and formulate responses, the impact of higher levels of experience can't be overlooked.

For classes where students have not had such experience, we strongly recommend that this course not be the first course in which students encounter a team-based

project. For such students it would be very helpful for this course to follow one or more courses that include a team-oriented project with a real customer, using versioning and change management tools. Otherwise, students are likely to thrash, struggling to learn too many things at once.

7.4 Availability of a Studio Project

We depend heavily on a substantial, year-long project with a real client. Such projects are richer and deeper than textbook projects, and the students have realistic interactions with clients, with all of the ambiguity, frustration, and need for diplomacy that such encounters require. Unfortunately, not all contexts in which this course might be taught will have access to such a project.

We have developed one alternative for our distance education version of the course. During the semester when the class sessions were recorded, we planned carefully to capture as much as possible of the contextual material for one of the studio projects. We have materials such as a videotaped initial presentation by the client of what they want, and both documentation and code from the client from the application of which the project will become a part. The goal is to provide enough material for the distance education instructor to act as a realistic surrogate client, answering questions and posing convincingly as an interviewee for contextual design.

One company that uses our distance offerings would like to use its own in-house project as a basis for student assignments. This would be a natural fit for this course, which would provide a different set of techniques from the company's usual process. We would need to be careful to avoid conflicts of interest between the course and the project goals, and we, by policy, do not offer course credit for work done on the job.

Another scenario for handling the lack of a real project is to create a mock-up project with sufficient problem complexity. This project can serve as a resource for typical issues that the students would encounter in a real project. The challenge of this approach is being able to introduce multiple real world constraints to the problem such as conflicting requirements, possible scenarios of integration with existing products, business and economic consideration, and non-trivial design issues. In addition, this approach requires the creation of substantial supplemental material that will be instrumental in introducing the project to the students. In this scenario the instructor needs to serve as the client as well, creating realistic challenges that may not necessarily happen in a course setting, but could occur in a real project. Carnegie Mellon West uses this approach in their Management of Software Systems Development program, where they have crafted a project in which they conduct the activities of the program as a series of mock-up client-employee interactions [12].

An alternative that would require a significant investment, but promises very broad and substantial benefits, would be to create a national resource in the form of one or more sample full records of software development projects. In addition to all versions of the code, maintained in a version control system, the resource should include a complete change history, records of design discussions, and inputs from customers and users. Although this idea has been discussed several times in the past, it has always stumbled on the question of acquiring a code base that can be made public. With the advent of open source, this problem has largely disappeared. Open source projects could provide the foundation for the resource we have in mind.

In order to create the sort of sample project records we have in mind, an open source project should be augmented with materials such as requirements and design documentation, contextual interviews with users, competitive analyses with other products, and training materials that instructors and teaching assistants can use to become convincing surrogate customers and users. The full project record could become the baseline in a version control system; each course could create its own code branch, allowing students to implement changes without burdening the open source project and without forcing students to cope with a code base that is changing underneath them (unless this serves an educational objective for a particular course).

In addition to serving as a resource for this particular class, such a resource would serve a variety of educational and research objectives. Researchers could, for example, directly compare the ability of various formal approaches such as model checking to detect significant bugs. Various ways of approaching important tasks such as refactoring could be directly compared. Visualization techniques could be tried out to see what they reveal about the code base that is relevant for various tasks. More generally, it would provide a very realistic example for many educational purposes, since our students are much more likely to spend their professional careers dealing with existing code bases rather than green field development.

8 Conclusion

The core problems of software engineering have evolved dramatically over the last decade or so, and we need to respond to these changes in order to equip our students with the skills and knowledge they need to excel in this new environment. The particular challenge for educators is that the need for the skills we taught ten or twenty years ago has not gone away. Students who find themselves working on embedded software, for example, still need to understand how to optimize for space and/or time, how to efficiently use machine-level instruction sets, memory overlays, and all the other skills that were important in the early days of computing. All our students still need to understand how to structure programs, how to handle exceptional conditions and errors, and how to organize the code so that anticipated changes can be localized. The need for basic skills in the design and construction of software, knowledge of how to make appropriate uses of abstraction and formality, have not gone away – in fact, as systems get larger, more critical, and more complex, the skills we have taught for the last 10 or 20 years are increasingly important.

Nevertheless, we believe that the technical, social, and business context of current developments provide additional challenges for which our traditional courses do not adequately prepare our students. Further, we believe that these additional challenges are sufficiently important to justify space in the curriculum. The course we describe in this paper represents our attempt to respond to the contemporary environment in a way that reinforces principles that underlie traditional teachings, and prepares our students to continue to learn, while providing skills they can apply immediately. The challenge to us as educators is to solve the curriculum design problem in a way that allocates the students' scarce attention to tasks that serve multiple goals at once. In an environment that imposes new challenges without relinquishing the old, that represents our only real hope of keeping up.

Acknowledgements

Thanks to colleagues, especially members of Carnegie Mellon's Institute for Software Research, International (ISRI) and its Master of Software Engineering (MSE) program, who have taught us about education, showed us new alternatives, and otherwise stimulated our appreciation of the problems and opportunities in software engineering education. Thanks especially to In-Young Ko, and Ho-Jin Choi for working with us on course development, and to all the students of the first four offerings of the course who helped us work out the ideas and the wrinkles of implementation. Sections 2 and 4 are derived from an ISRI working paper on curriculum design [38], and we particularly thank Jonathan Aldrich, Ray Bareiss, Shawn Butler, Lynn Carter, Owen Cheng, Steve Cross, Jamie Dinkelacker, Dave Farber, David Garlan, John Grasso, Martin Griss, Tim Halloran, Carol Hoover, Lisa Jacinto, Mark Klein, Deniz Lanyi, Beth Latronico, Jim Morris, Priya Narasimhan, Joe Newcomer, Linda Northrup, Mark Paulk, Mel Rosso-Llopart, Walt Shearer, Bill Scherlis, Todd Sedano, Gil Taran, Jim Tomayko, and Tony Wasserman for their contributions. Portions of Section 1 are derived from Shaw's education roadmap [37]. Carnegie Mellon does not have a definitive statement of the Carnegie Plan. Minor revisions are regularly made for different settings; the version presented here is a close variant of versions that have appeared in University publications over the years.

References

1. Omer Akin. *The Psychology of Architectural Design.* Pion, 1987.
2. Frank Armour and Granville Miller: *Advanced Use Case Modeling: Software Systems.* Addison-Wesley, 2001.
3. Toru Asada, Roy F. Swonger, Nadine Bounds, and Paul Duerig. The Quantified Design Space. In Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall 1996, Sec 5.2, pp 116-128.
4. Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufman, 1998.
5. Barry W. Boehm et al. *Software Cost Estimation with COCOMO II*. Prentice Hall 2000.
6. Barry W. Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer,* January 2001, pp. 2-4.
7. Benjamin S. Bloom and David R. Krathwohl (ed). *Taxonomy of educational objectives: The classification of educational goals. Handbook I, cognitive domain*. Longmans, Green, 1956.
8. John Seely Brown and Paul Duguid. *The Social Life of Information*. Harvard Business School Press, 2000.
9. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. 20th Anniversary Edition, Addison-Wesley Professional, 1995.
10. Carnegie Mellon University. Distance Education in Software Engineering. Degree program description, Carnegie Mellon University, `http://www.distance.cmu.edu/`
11. Carnegie Mellon University. *Master of Software Engineering*. Degree program description, Carnegie Mellon University, February 2005, `http://www.mse.cs.cmu.edu/`

12. Carnegie Mellon West. *Management of Software Systems Development*. Degree program description, Carnegie Mellon University, February 2005,
13. John M. Carrol. *Making Use: Scenario-Based Design of Human-Computer Interactions*. MIT Press, 2000.
14. Clayton Christensen. *The Innovator's Dilemma.* Harper Business, 2000.
15. Computer Science and Telecommunications Board, National Research Council. *More than Screen Deep: Toward Every-Citizen Interfaces to the National Information Infrastructure.* National Academy Press, 1997.
16. Computer Science and Telecommunications Board, National Research Council. *The Digital Dilemma. Intellectual Property in the Information Age.* National Academy Press, 2000.
17. Computer Science and Telecommunications Board, National Research Council. *Global Networks and Local Values.* National Academy Press, 2001.
18. Michael Cusumano and David Yoffie. *Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft.* Touchstone, 1998.
19. Thomas H. Davenport. The case of the soft software proposal. *Harvard Business Review*, May-June 1989.
20. Daniel M. Hoffman and David M. Weiss (eds.). *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.
21. IEEE Computer Society Professional Practices Committee. *SWEBOK: Guide to the Software Engineering Body of Knowledge, 2004 version*. IEEE Computer Society, 2004.
22. Michael Jackson. *Problem Frames.* Addison-Wesley, 2001.
23. 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. ACM and IEEE Computer Society, August 2004.
24. Tracy Kidder. *Soul of a New Machine.* Back Bay Books, 2000.
25. L. Korba and S. Kenny. Towards Meeting the Privacy Challenge: Adapting DRM. *ACM Workshop on Digital Rights Management*, Washington, DC, November 2002.
26. Lawrence Lessig. *The Future of Ideas.* Random House, 2001.
27. The Localization Industry Standards Association (LISA). *The Localization Industry Primer*, 2nd edition. LISA, 2003.
28. Fred Moody. *I Sing the Body Electronic: A Year with Microsoft on the Multimedia Frontier.* Penguin Books, 1995.
29. Donald Norman. *The Design of Everyday Things.* Currency/Doubleday, 1990.
30. Charles Perrow. *Normal Accidents.* Princeton University Press, 1999 (updated edition).
31. Henry Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering.* Cambridge University Press, 1994.
32. Vahe Poladian, David Garlan, and Mary Shaw. Software Selection and Configuration in Mobile Environments: A Utility-Based Approach. *Position paper for the Fourth Workshop on Economics-Driven Software Engineering Research (EDSER-4),* affiliated with the 24th International Conference on Software Engineering (ICSE'02), May 2002.
33. Vahe Poladian, Shawn A. Butler, Mary Shaw, David Garlan. Time is Not Money: The Case for Multi-dimensional Accounting in Value-based Software Engineering. Position paper for the *Fifth Workshop on Economics-Driven Software Research (EDSER-5*), affiliated with the 25th International Conference on Software Engineering (ICSE'03), May 2003.
34. Reidenberg, J.R. Resolving conflicting international data privacy rules in cyberspace. *Stanford Law Review* 52 (2000), pp. 1315-1376.

35. Carl Shapiro and Hal R. Varian. *Information Rules: A strategic guide to the network economy.* Harvard Business School Press, 1998.
36. Mary Shaw (ed). *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Springer-Verlag, 1985, 198 pp.
37. Mary Shaw. Software Engineering Education: A Roadmap. In A. Finkelstein (ed), *The Future of Software Engineering*, pp. 371-380), ACM Press, 2000.
38. Mary Shaw (ed). Software Engineering for the 21st Century; a basis for rethinking the curriculum. *Technical Report CMU-ISRI-05-108*, Institute for Software Research International, Carnegie Mellon University, March 2005.
39. Mary Shaw, Ashish Arora, Shawn Butler, and Chirs Scaffidi. *In search of a unified theory for early predictive design evaluation.* Working paper, 2004.
40. Mary Shaw, Jim Herbsleb and Ipek Ozkaya. *Methods: Deciding What to Design*. Course home page. Sept. 2005- December 2005. School of Computer Science, Carnegie-Mellon University. `http://spoke.compose.cs.cmu.edu/method-fall-05/`
41. Herbert Simon. *The Sciences of the Artificial*. MIT Press, 1996 (3rd edition).
42. Herbert A. Simon. What we know about learning. *Journal of Engineering Education*, vol 87 no 4, Oct 1998, pp. 343-348. Cached at `http://www.ic.polyu.edu.hk/esh/KB/good-practices/GTP9016.pdf`
43. Carolyn Snyder. *Paper Prototyping*. Morgan Kaufman, 2003.
44. Lucy Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication.* Cambridge University Press, 1987.
45. Walter Vincenti. *What Engineers Know and How They Know It.* Johns Hopkins University Press, 1990.
46. Eric von Hippel. *The Sources of Innovation.* Oxford University Press, 1994.
47. M. Mitchell Waldrop. *The Dream Machine: J.C.R. Licklider and the Revolution that Made Computing Personal.* Penguin Books, 2001.
48. Terry Winograd (ed). *Bringing Design to Software*. Addison-Wesley, 1996.

Appendix. The Carnegie Plan for Engineering Education

A Carnegie Mellon education aims to prepare students for life and leadership. In a continually changing world, the most important qualities we can help our students develop are the ability to think independently and critically, the ability to learn, and the ability to change and grow. As future leaders they must have courage to act, be sensitive to the needs and feelings of others, understand and value diversity, and honor the responsibilities that come with specialized knowledge and power.

Carnegie Mellon's educational programs are designed to help students acquire:

- *Depth of knowledge* in their chosen areas of specialization and *genuine intellectual breadth* in other fields.
- *Creativity and intellectual playfulness*, moving beyond established knowledge and practice to create imaginative ideas and artifacts.
- *Skilled thoughtfulness and critical judgment*, which allow them to evaluate new ideas; identify and solve or explore problems; and appreciate a variety of different forms of analysis and thought.
- *Skills of independent learning*, which enable them to grow in wisdom and keep abreast of changing knowledge and problems in their profession and the world.
- *A considered set of values*, including commitment to personal excellence and intellectual adventure, a concern for the freedoms and dignity of others, and sensitivity to the special professional and social responsibilities that come with advanced learning and positions of leadership.
- *The self-confidence and resourcefulness* necessary to take action and get things done.
- The *ability to communicate with others* on topics both within and outside their chosen field of specialization.

Most instruction at Carnegie Mellon is focused on fundamentals useful in later learning, rather than on particulars of knowledge and techniques, which may soon become obsolete. Advanced courses provide students with the opportunity to refine their skills by applying and exercising the fundamentals they have acquired in earlier courses and by exploring new analytical and creative directions. We are committed to bring together the traditions of liberal and professional education. In a world which has sometimes placed too little emphasis on "skill," we take pride in educating students who display excellence in application, students who can do useful things with their learning.

Values, including a sensitivity to the feelings, needs, and rights of others, are learned in part through example. To this end, the faculty and staff of Carnegie Mellon work to provide a supportive and caring environment that values and respects intellectual, philosophical, personal, and cultural diversity. The faculty strive to identify and discuss with their students, both in formal classroom settings and in a variety of informal contexts, their responsibilities as professionals, citizens and human beings, and to teach through example.

The educational programs at Carnegie Mellon are designed to help our students become accomplished professionals who are broadly educated, independent, and humane leaders.