DESIGN IN E-GOVERNMENT CURRICULA: BOTTOM-UP VS. TOP-DOWN

2nd Scandinavian E-Government Workshop, February 2005. Revised version of Feb 2, 2005.

Niels Jørgensen

Roskilde University, Computer Science Department, Building 42.1, Universitetsvej 1, 4000 Roskilde, Denmark.
Email: nielsj@ruc.dk.

Abstract

Design should be a key paradigm in e-government curricula, because e-government is the design of new administrative, organizational and technical systems. For the technical aspects of e-government, a design perspective may guide the identification of topics that a curriculum should cover, such as software architecture, and the selection or development of approaches to teaching them. A survey of recently published ACM and IEEE curriculum reports indicates that their main approach to the teaching of technical design is *bottom-up.* In a bottom-up approach, the subsystems of a system are studied as a prerequisite for understanding the design of the system as a whole. Two alternative approaches are sketched: *generic design* and *top-down design*. While the latter approaches are less prevalent and perhaps less well understood, they deserve further exploration because they may provide the only realistic road for students to ever reach the point of studying the level of an e-government (or other) system as a whole.

Introduction

At a workshop on formal education in e-government held at Roskilde University in April 2004 (Andersen et al., 2004), representatives from government and software vendors said almost equivocally that the graduates they sought should possess business and organizational competencies rather than technical. This was a disturbing message to the university representatives, especially those affiliated with Computer Science departments, including the author. The message could potentially reinforce students in avoiding technical topics ('I study to become a project manager, not a programmer'), and seemed to lack critical reflection on current practices of staffing e-government projects with graduates of Economics, Law and other disciplines that do not provide technical insight.

Rather than giving precedence either to technical or non-technical topics, a crucial consideration for educators is the development of curricula that provide insight into both. This entails a focus on time-conserving or minimalist approaches to technical topics, in order to attain a balance of topics in the over-all curriculum. Design can be a key concept in such 'optimized' approaches to the teaching of technical topics, and a link to organizational and administrative topics. In the sequel, design is viewed as technical system design, rather than aesthetic or user-interface design; this view can be summarized as follows:

> *Design is the definition of a system in terms of a of its subsystems, with the purpose of meeting specific use-related goals, and subject to technical, economic and other constraints.*

More specifically, a design perspective may provide guidance in the identification of topics to be covered by the technical part of a curriculum. It implies priority to topics such as software architecture (eg. connecting subsystems by means of the Internet) and security (eg. digital

signatures), because these are of relevance to the over-all design of e-government systems. Indeed, much attention is devoted to security and architecture in recent Danish government white papers on e-government, for example the landmark report (Finance Ministry, 2001).

A major difficulty associated with teaching of system-level design of e-government systems is their complexity. To shed light on this difficulty, a series of recommended curricula for Computer Science and related disciplines published recently by ACM and IEEE is surveyed. (References are provided below in the section The ACM and IEEE curriculum reports.) The series is of interest because several of the reports are highly focused on design. Moreover, the series comprises four distinct fields and so provides a potential for the comparison of different approaches. Finally, the reports are interesting because they recommend that students also study an application domain - in other words, e-government for example !

The conclusion of the survey is that the prevailing approach to the teaching of design in the ACM and IEEE reports is a bottom-up approach. Among other things this is based on the observation that all four reports prioritize teaching the same low-level subsystems, for example knowledge of operating systems and skills in programming-in-the small, eg. of small Java programs. Also, despite many insightful and visionary general comments in the reports, the concrete approach to the teaching of design is implicit, in the sense of not motivated explicitly.

In an attempt to clarify the rationale underlying the surveyed reports, and also to provide a basis for a discussion of alternative approaches, the three idealized or archetypal approaches to the teaching of design, *bottom-up*, *generic*, and *top-down*, are examined.

In a wider perspective, a clarification of advantages of various approaches to the teaching of design within formal education programs may also be of interest from a perspective of enhancing life-long learning, and a perspective of gaining insight into the design process per se.

The paper is organized as follows: Following a brief sketch of the paper's method, the first part of the paper surveys the four ACM and IEEE reports. The second part of the paper identifies and examines the approaches termed bottom-up, generic, and top-down to the teaching of design. A final section concludes.

Method

The survey in the first part of the paper attempts to identify different approaches in the ACM and IEEE curriculum reports to the teaching of design. This is done by examining the reports with the aim of answering two questions: What role is assigned to design in the general discussion of the discipline ? and how is this implemented in the concrete curriculum ?

The sketch in the second part of the paper of three approaches to the teaching of design is based on the definition given above of design in terms of the system/subsystem distinction. The definition entails a view of layered systems that can be traversed either bottom-up or top-down, and thus suggests a coarse grouping of design approaches. The top-down, bottom-up, and generic approaches are all present in the ACM and IEEE reports, with bottom-up prevailing. An attempt is made to list relevant arguments in favor of all three approaches. Arguments are sought in the reports themselves and in literature on Computer Science education. It is not the intension to evaluate the three approaches, except for noting the obvious risks of bottom-up and top-down, namely that students never advance to design issues at a sufficiently high or low system level.

The ACM and IEEE curriculum reports

The recent curriculum revision initiative undertaken jointly by ACM and IEEE began in 1998 and led to the publication of four curriculum recommendations through 2001-2004. The publications and their common abbreviations, which will be used in the sequel, are:

> *Computer Science (Joint Task Force on Computing Curricula, 2001): CC 2001.*
>
> *Software Engineering (Joint Task Force on Computing Curricula, 2004): SE 2004.*
>
> *Information Systems (J. T. Gorgone et al., 2002): IS 2002.*
>
> *Computer Engineering (Joint Task Force on Computer Engineering Curricula, 2002): CE 2002.*

Each report defines a curriculum for a four year undergraduate program. It is assumed that the typical student is also studying an area or discipline outside of computing, yet spending the most time on the computing discipline.

There is some inconsistency in the way the reports use the term *computing*. Each discipline report starts from scratch by defining the role played by the discipline within computing, where computing is seen as the broader area whose various parts are covered by the disciplines. However, the Computer Science report is called CC 2001, where CC abbreviates *computing curriculum*. It would of course have been more logical to label the Computer Science report CS, and reserve CC for a publication about the entire computing revision effort. This inconsistency may reflect a view of Computer Science as the most central of the four disciplines.

In the sequel, each of the four reports are analyzed in turn. The findings regarding the role played by design is summarized in Table 2 below.

Computer Science (CC 2001)

The Computer Science report (CC 2001) is in many ways visionary. In particular, it provides an interesting characterization of Computer Science as a *research* field, advocates a sort of principles-first approach, and discusses weaknesses of prevailing curriculum approaches, including programming-first and technology-orientation. Weaknesses of a programming-first approach cited in CC 2001 include giving students a limited sense of the discipline as equal to programming, and of programming as equal to programming-in-the-small and mastery of language details. The risk of technology orientation is exemplified by compiler or operating system courses that are "system-artifact dinosaurs" (CC 2001, p 35, quoting from Shaw, 92), where presumably students are overwhelmed by details about large if not extinct software systems.

A previous curriculum initiative led to the publication of a computing or Computer Science curriculum in 1989-1991 (Denning et al., 1989), and there are two even earlier predecessors from 1968 and 78, respectively. The 1989 report is even more ambitious that its 2001 successor, in the sense of reflecting even more over the field of Computer Science, and since the latter reports cites the former for background, it seems permissible to do so in the present paper as well.

Meta-level definition of Computer Science in CC 2001. The definition of Computer Science in (Denning et al., 1989) and CC 2001 as a research field comprises two levels. The first level is a

kind of meta-level, or philosophy of science level: Computer science is said to be distinguished from the natural sciences by the way it is related to engineering: Computer Science is *integrated* with engineering, where the natural sciences are *separated* from the corresponding engineering disciplines. For example, natural sciences such as physics and chemistry are separate disciplines from engineering disciplines such as civil and chemical engineering.

At this general level, three fundamental paradigms of Computer Science are identified: theory, abstraction, and design. These paradigms are said to be of equal importance (CC 2001, p 36, and Denning et al, 1989, p 10). The way computing integrates science and engineering is expressed in terms of the three paradigms: Theory and abstraction correspond to science, and abstraction and design to engineering (see Figure 1).

Design is defined as follows:

> *"[Design is] rooted in engineering and consists of four steps followed in the construction of a system (or device) to solve a given problem: (1) state requirements; (2) state specifications; (3) design and implement the system; (4) test the system" (Denning et al., 1989, p 10).*

The theory paradigm is viewed as a form of applied mathematics, for example complexity theory, with formal theorems and proofs.

Abstraction is described as a broad paradigm that includes modeling and experimentation. These are connected in a straightforward manner: abstraction is viewed as a prerequisite for modeling, and modeling as providing guidance to the construction of experiments; and experiments are said to be useful to predict the behavior of future implementations, and thus related to design. All in all, at the general level, Computer Science is defined with great emphasis on design; design represents the engineering component of Computer Science, and is understood as the full range of creative and constructive activities during the software life cycle.

Concrete-level definition of Computer Science in CC 2001. The second level in the definition of Computer Science is a concrete level, where Computer Science is defined in terms of a body of knowledge. The knowledge body is defined as a hierarchy. The body consists of fourteen core knowledge areas (cf. Figure 1) which are subdivided into units; in turn units are divided into

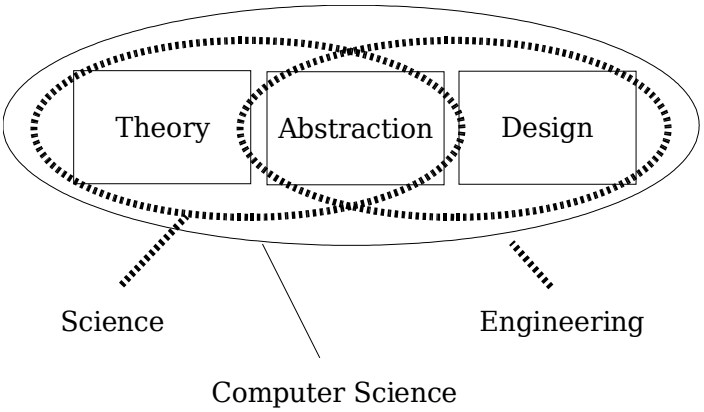

Figure 1. Computer science comprises theory, abstraction, and design, and so integrates areas traditionally studied in science and engineering, respectively, according to the CC 2001 report.

topics. For example, knowledge area Algorithms and Complexity contains knowledge unit Cryptographic algorithms (and many others), which contains topic digital signatures (and many others); and knowledge area Programming Languages contains knowledge unit Abstraction mechanisms with topic Modules in programming languages.

It is note-worthy that even at the concrete level, the definition of Computer Science is given from a research point of view. Knowledge areas were defined in (Denning et al., 1989) in terms of a requirement that they possess: (1) underlying unity of subject matter, (2) substantial theoretical component, (3) significant abstractions, and (4) important design and implementation issues. In other words, knowledge areas were defined with reference to the three basic paradigms. There is also a criterion of a sociological nature: knowledge areas should be based on research communities having their own literature.

Courses are defined only in a next step, namely in terms of what parts of knowledge areas they cover. Many courses cover material from several knowledge areas, for example an introductory course may cover units from Algorithms and Complexity as well as Programming Languages.

The CC 2001 report aims at proposing an up-to-date Computer Science curriculum in the sense of covering new phenomena, such as the world wide web, or phenomena of increased significance, such as object oriented programming. The growth of the discipline is reflected in an increasing number of knowledge areas, ie. from nine in the 1989 report (Denning et al., 1989) to 14 in CC 2001. It is interesting, though, that while CC 2001 speaks rather eloquently about the rapid emergence of new topics, the curriculum devotes more time to Assembly Level machine organization than to Human-Computer Interaction (p 85).

Characterization of the design approach in CC 2001. In several ways the approach to design in CC 2001 is a *systems development-oriented* approach. Such is the approach in the description of the Software Engineering knowledge area, which is one of the five largest areas and defined in terms of the units software design, software processes and software evolution, and others. This approach can also be seen in two other contexts:

The first context is the explicit definition of a set of approaches to the intermediate level of the undergraduate program, in the form of a set of courses. One of these is the systems-based approach (CC 2001, p 38) which focuses on systems development. In this approach, each course combines topics from Software Engineering with topics from other knowledge areas, for example topic object oriented programming of Programming Languages, under an over-all perspective of software development.

The second way that CC 2001 approaches design from a system development-perspective is by strongly suggesting that students at the end of an undergraduate program follow a one or two semester course where team of students work on a so-called capstone project, that is, develop a software product of more considerable size than would be possible as part of fulfillment of weakly assignments etc.

Despite the fact that the focus on software development implies some emphasis on design, that emphasis does not appear to fully match the hailing of design as one of three fundamental paradigms, in the meta-level definition of Computer Science. Moreover, since both the systems-approach to courses and the capstone project are labeled in CC 2001 as optional, the system development-based approach to design in CC 2001 to design is labeled *optional* in Table 2.

Fortunately, the 1989 Computer Science report provides some compensation for the vagueness of the design-related guidelines in CC 2001. Recall that the 1989 report is the source of the definition in CC 2001 of Computer Science in terms of theory, abstraction, and design. In the 1989 report, a list of theory, abstraction, and design examples were given for each knowledge area. For example, the list suggested for knowledge area Programming Languages included (Denning et al., 1989, p 18):

> *Theory: Turing machines; formal semantics of programming languages.*
>
> *Abstraction: Classification of programming languages into eg. object oriented and procedural; models for program structure; implementation models.*
>
> *Design: specific programming languages; specific implementations.*

It may be noted that the design examples listed above are artifacts, ie. textually specified artifacts (programming languages) and software artifacts (implementations). Indeed, most design examples listed in (Denning et al., 1989) are software or other artifacts. For example, design examples in knowledge area Algorithms and Data Structures includes cryptographic protocols and algorithms for important problems, both of which are textually defined artifacts, and design examples in knowledge area Operating Systems includes network protocols such as TCP, etc. By exception, knowledge area Software Engineering is described in part by reference to proper design methods; however, most examples even for Software Engineering are artifacts: namely software development tools such as specification languages, debuggers, etc. All in all, only a small part of the design examples would qualify as a method or technique that is in some way part of design activities, even in the broad sense of design as construction of systems from specification to test.

Therefore, the design paradigm in (Denning et al., 1989), and by implication i CC 2001, appears to be represented by a wealth of artifacts. The typical artifact is a representative of some principle, for example a language class, the idea being that a given course should pick some class instance, eg. Java. It seems reasonable to term this approach as the *illustrative artifacts* approach to teaching design, and moreover, as an *implicit* approach, because there is no discussion of how this exposure to artifacts may actually serve to illustrate design.

Software Engineering (SE 2004)

The Software Engineering report, SE 2004, defines a full undergraduate curriculum, analogously to the three other discipline reports. The discipline is defined with historical reference to the 1968 NATO Conference on Software Engineering (Naur et al., 1969) and the complexity of software as discussed by Brook (Brooks, 1995). As in CC 2001, the discipline is defined partly in terms of knowledge areas. In contrast to CC 2001, the discipline is defined as a professional field, rather than as a research field. Knowledge areas are called Software Engineering *education* knowledge areas, and motivated by practical relevance. A large knowledge area is Computing Essentials, which overlaps with several knowledge areas of CC 2001, and contains material considered as important prerequisites for the professional software engineer.

The centrality of design. Two principles are stressed in the general definition of Software Engineering in SE 2004: First, the centrality of the concept of design; and second, the precedence of Computer Science knowledge over 'process'. The centrality of design is argued on

the grounds that design is central to any engineering field. In this general context of engineering, design is "*the definition of a new artifact by finding technical solutions to specific practical issues, while taking into account economic, legal, and social considerations*". Design and engineering at this general level is seen as involving, among other things, trade-off analysis, decision making, team work, and experiments and measurements.

Software engineering is seen as comprising implementation and testing activities, but unlike the definition provided in CC 2001 these activities are not seen as part of design, which provides (merely) "prerequisites for the physical realization of a solution". This is consistent with definitions of design in eg. civil engineering, where the design of a bridge is expressed in blueprints and other prerequisites for the physical construction process, while not actually comprising the latter process.

Precedence of Computer Science over 'process'. The downplay of the role of process is stated as follows: "A common misconception about Software Engineering is that it is primarily about process-oriented activities". These are exemplified as "requirements, design, quality assurance, process improvement, and project management" (SE 2004, p 6). These critical remarks appear as being directed not against a 'soft' or Information Systems approach, but against over-emphasis on process as such, hard or soft. Process is important but not at the expense of the Computer Science context - this seems to be the point. The report lists a set of properties of Software Engineering that distinguish it from "more traditional fields of engineering" (SE 2004, p7): Software engineering deals with intangible and logical (rather than physical) artifacts; there is no manufacturing process; there is a basis in discrete (rather than continuous) mathematics; and maintenance can be seen as continued development (rather than dealing with wear and tear). As an aside it may be noted that dealing with intangible artifacts could be seen as the most fundamental property in the sense that it implies the others.

Characterization of the design approach in SE 2004. The approach in SE 2004 to the teaching of design may be characterized as *technical design*. Design, emphasized as central, is integrated or 'buried' in a Computer Science context, consistently with the downplay of generic process. In the concrete definition of Software Engineering given in SE 2004 in the form of knowledge areas, there is no single knowledge area capturing the "essentials of design in Software Engineering"; rather, design issues are part of all ten knowledge areas. For example, the units that knowledge area Software Design are composed of are mainly technical; they include unit Architectural Design, which includes, in turn, a topic named Architectural Styles. The topic is defined by a list of example styles, which are all complex technical concepts, for example pipes, filters, layers, transactions, and events. The complexity of these concepts can also be seen from the fact that the courses defined to cover them in SE 2004, for example the course SE 311 (Software Design and architecture, SE 2004, p 107), require several technical courses as prerequisites.

Another aspect of the 'technical' approach to design is the focus on quantitative methods. Knowledge area Mathematical and Engineering Fundamentals comprises, among others, a number of quantitatively oriented units, such as statistical analysis and theory of measurement. This approach is reflected in several general and programmatic statements such as "software development practice [needs the] rigor that the engineering disciplines bring to the reliability and trustworthiness of the artifacts they engineer" - by the way, statements that would seem to emphasize generic process, rather than downplay it.

The context-dependency of design is emphasized even further: The Software Engineering report stipulates that graduates should "come to terms with at least one application domain". The report makes a statement (SE 2004 p 9) that Software Engineering has been the most successful within specific application domains. One argument given for this is that for engineers to be able to distinguish standard parts from those that must be developed from scratch, they must have domain specific knowledge. The report recommends that institutions experiment with the development of undergraduate programs that integrate Software Engineering into application domains, and list for example "Aerospace Software Engineering". A program in "E-government Software Engineering" would fit in here as well! Therefore, the approach to design may be characterized also as *domain-dependent* design.

Information Systems (IS 2002)

The IS 2002 report defines Information Systems with reference to a job function: the "Information Systems organization function" (IS 2002, p10). Academic programs in Information Systems are viewed as analogous to business school programs that correspond to organizational functions such as management of financial or human resources. More specifically, the Information Systems function in an organization is defined as "acquisition, deployment, and management of it resources and services". Additionally, the IS field is defined with reference to activities of "system development, system operation, and system maintenance", that is, activities that are typical of the software vendor, and so extend beyond the purchasing organization.

Discipline	***Perceived role of design in discipline***	***Definition of design***	***General recommendation***	***Actual approach***
Computer Science (CC 2001) (Denning et al., 1989)	Design, theory, and abstraction are the three fundamental paradigms.	Construction of system from specification to test	Avoid artifact-centric courses Avoid identification of Computer Science with programming	Systems development-oriented (optional) Illustrative artifacts (implicit)
Software Engineering (SE 2004)	Design is the fundamental paradigm in engineering practice.	Definition of a new artifact by finding technical solutions to specific practical issues, while taking into account [..] considerations	Avoid process-orientation Emphasize rigor of engineering approach	Technical design Domain-dependent design
Information Systems (IS 2002)	No particular role.	No definition.	None	Conventional (implicit).
Computer Engineering (CE 2002)	Same as SE 2004	Same as SE 2004, though emphasis on choices, trade-offs, and constraints.	Design must pervade entire curriculum	Students' own work on design assignments.

Table 2. The role of design in the ACM and IEEE curriculum reports.

The IS curriculum is motivated, at a general level, by a characterization of the competencies that IS professionals should possess: sound technical knowledge, understanding of organizations, and understanding of achievement of organizational goals with information technology (p11).

'**Exit characteristics**': Similarly to the Software Engineering report, the IS report views its field as a practical field; however, while the former characterizes Software Engineering in terms of the types of tasks engineers solve and how they solve them, the IS report merely state a set of required competencies of the graduate. For example, it is striking that the IS 2002 curriculum is *not* motivated by a discussion of current professional challenges, such as challenges related to organizational change and the need for practitioners that *integrate* competencies of both a technical and business/organizational nature. Instead, the curriculum is defined with reference to a set of somewhat arbitrary, so-called 'exit characteristics" of IS graduates. These are stated in highly general terms such as "Be problem solvers and critical thinkers" (p6). Another apparent omission at the level of the general definition of the field of IS is that there is no discussion of the role of design. This is striking because a discussion, for example, of the interplay between technical and organizational/administrative design could provide motivation for IS as an independent academic program.

The need for prioritizing of topics is perhaps even larger in the definition of a curriculum for Information Systems than for Computer Science or Software Engineering, because the former contains larger portions of both of the others, in addition to business and organizational material. However, the report does not discuss criteria for selecting topics or knowledge areas that are the most relevant as 'exit characteristics' of IS professionals.

The concrete curriculum definition in IS 2002 is given in the form of ten suggested courses. The courses are defined at a detailed level with topics and learning goals, and prerequisites are shown in a course graph. The courses are grouped into five learnings areas. Learning area "Information Technology" contains three courses, indicating a strong emphasis on technical material rooted in Computer Science, for example in the course "Networks and Telecommunication". Learning area "Information Systems Development" also contains three courses, and design is central in all three; the terminology in IS 2002 includes "logical design" and "physical design", corresponding to system specification and design. Learning area "IS Fundamentals" contains two courses each of which approach the interplay between organization and technology. For example, the course "Electronic Business Strategy, Architecture and Design" focuses on "the linkage between organizational strategy and networked information technologies".

Characterization of the design approach in IS 2002. The report's approach to design can be characterized as *conventional* and *implicit*. The approach is conventional in the sense that the report stresses that students must learn to construct databases and programs, which is similar to the approach in the other discipline reports. For example, there is a statement that "*Instruction in physical design of Information Systems will ensure that the students can use a logical design to implement Information Systems in both a DBMS and in emerging development environments*", the latter referring to competencies in programming with object oriented and procedural languages. The term conventional is meant to indicate that in the concrete curriculum, design plays a similar to the other disciplines. Finally, the approach can be said to be implicit because there is no discussion of the role of design. IS 2002 in a way is the most consistent with respect to design emphasis, in that its general remarks do not raise false expectations.

Computer engineering (CE 2002)

The last of the four disciplines targeted by the ACM and IEEE curriculum revision effort is Computer Engineering. It is included in this survey not only for completeness, but also because the report adds to a picture of all four suggested curricula being rather similar. One wonders: shouldn't there be more difference between eg. Computer Engineering and Information Systems ?

At a general level, CE 2002 provides an informal definition of Computer Engineering as a combination of Computer Science and Electrical Engineering. The origin of the development of Computer Engineering into a discipline separated from Electrical Engineering is said to be the widespread diffusion in current society of embedded systems: small or large systems ranging from mobile phones to automobiles equipped with full CPUs and associated memory and other devices - rather than conventional electronic devices of the sort studied in Electrical Engineering.

Computer Engineering is defined as a professional field, analogously to the Software Engineering and Informations Systems fields. There are many similarities with the Software Engineering report, most notably the emphasis on design. The CE 2002 report lists the "ability to design computer systems that include both hardware and software" as one of three characteristics of computer engineers, the two others being breadth of mathematical and engineering knowledge, and preparation for professional practice.

The concrete curriculum contains knowledge areas such as Digital Signal Processing and Circuits and Signals. These technical areas testify to the emphasis on hardware which distinguishes the Computer Engineering curriculum from the three other programs. Knowledge area Computer Systems Engineering may capture the essence of the discipline, with its focus on "the development of new devices such as digital cameras, hand-held computers, .." (CE 2002, Appendix A, p6). The description of this knowledge area focuses on design and may contain the richest account, in all four discipline reports, of the activities involved in design: there is focus on errors, on strength and weaknesses of solutions to design problems, and emphasis on trade-offs and choices, such as whether to implementing functionality in software or hardware.

Characterization of the design approach in CE 2002. The report suggests that students work on a large project towards the end of the study program analogously to the capstone projects in Computer Science and Software Engineering, and design is suggested as the primary theme for this project. Also, experience with development and test of various designs is suggested as a key theme for laboratory exercises. In stressing design as a theme for a final project as well as for laboratory work, CE 2002 can be said to be the curriculum that places the most emphasis on design. Since the report also stresses design as a key topic in course assignments, and to differentiate it from the Software Engineering report, the approach to design in the Computer Engineering report may be characterized as *student's own work on design assignments*.

Three design approaches: "Bottom-up", "generic", and "top-down"

In order to understand better the approach to the teaching of design in the surveyed reports (as summarized above in Table 2), this section tentatively defines the three approaches bottom-up, generic, and top-down, and list arguments that can be given in favor of each of them. The arguments are summarized below in Table 3. The approach in the surveyed reports is

characterized as mainly bottom-up, yet with some elements also of generic and top-down.

The purpose of the tentative construction of the three idealized or archetypal approaches to the teaching of design is to provide a basis for a discussion of alternatives to the approach that prevails in the ACM and IEEE reports.

The labeling of the design approaches as top-down vs. bottom-up is with respect to a metaphor of hierarchical systems, where a system is a composition of subsystems, and so on recursively. Given this view of systems, design can be defined as follows (same definition as on first page):

> *Design is the definition of a system in terms of a of its subsystems, with the purpose of meeting specific use-related goals, and subject to technical, economic and other constraints.*

The system/subsystem view entails a view of layered design, ie. divided into over-all system design, subsystem design, etc. This view is consistent with the software design principle of division of programs into subprograms, and prevailing views on the importance of related concepts such as data abstraction and information hiding.

The design definition given here is similar to the definition given in the Software Engineering and Computer Science reports (see subsections Computer Science and Software Engineering above), with two exceptions: First, those definitions do not refer to layers or subsystems. Second, the layered definition does not refer to *implementation* or *test*, only *definition* of system, whereby it is dissimilar to the definition given in CC 2001 (but similar to SE 2004). It is not the intension to exclude these activities from the design concept. Indeed, programming can be viewed as design at a concrete level, that is, as the definition of a small subsystem in the form of a source code file, and testing can be viewed as a method for design refinement.

Bottom-up design

The bottom-up approach to the teaching of design is defined as teaching students first and foremost about subsystems; design issues associated with aggregate system levels are taught later and with less emphasis.

This appears to be the prevailing approach in all four reports resulting from the ACM and IEEE curriculum initiative. Indeed, it is striking that despite the diversification into four distinct disciplines, the curricula are similar in that they all prioritize the same low-level subsystems, for example knowledge of operating systems and skills in programming-in-the small, eg. of small Java programs. The Software Engineering report is the most explicit in this respect, with its downplay of 'process' and emphasis of the discipline's 'roots' (SE 2004, p 7) in Computer Science.

Physicalism: A bottom-up approach bears some resemblance with an approach to civil engineering and natural science curricula sometimes termed physicalism (Voetmann, 2002), where physics is taught early in the curriculum because the discipline is considered foundational. An associated view may be that mathematics, physics, chemistry, biology, and engineering form a chain of dependent fields with eg. physics forming the basis for chemistry:

> *"Physics is the most fundamental of all natural sciences. Chemistry deals with .. the application of the laws of physics [..] Biology must lean [..] on physics and chemistry [..] The application of the principles of physics and chemistry to practical problems [..] has*

given rise to [..] engineering." (Alonso and Finn, 1980, p 6, quoted from Voetmann, 2002).

A parallel view of the subdisciplines of Computer Science may be expressed as follows in terms of layers of abstract machines. *To understand the design of applications at a given layer, understanding the underlying layer is a useful prerequisite, because the applications are implemented in that layer's language.* Example layers are machine instruction layer, operating system layer, and layers in network reference models, such as the OSI/ISO model.

Retrospective design: The approach in the Computer Science report was characterized as illustrative artifacts (cf. Table 2 above). In this approach, Java is a representative of object oriented programming languages, etc. From a design point of view, this can be seen as 'exposure to design results', where the design process is viewed retrospectively: An artifact, such as a programming language, is seen as the result of a design process that implements eg. object oriented principles and also embodies design trade-offs, weaknesses, etc.

Immaturity: Justification for a bottom-up approach may be found in Mary Shaw's work on Software Engineering and education. She argues in (Shaw, 1990) that Software Engineering is less mature than comparable disciplines such as civil and chemical engineering, rooted as they are in thousands of years of craftsmanship, and hundreds of years of organized production and scientific progress (in physics and chemistry). The main argument is based on the distinction between routine and innovative design problems. Routine problems are those that have occurred previously, and for which solutions have been developed which are known to work in practice. A science that supports an engineering practice should support the development of solutions to recurring practical problems, and codify solutions in a manner easily accessible by practitioners, for example in handbooks. Mature disciplines institutionalize mechanisms for maintaining and distributing handbook-type knowledge, for example *Perry's Chemical Engineering Handbook* (published by McGraw-Hill). Shaw noted in 1990 that the increasing focus on software reuse was a promising example of re-use of knowledge and solutions. However, her over-all finding was that in typical Software Engineering practice, design tasks were treated as posing novel problems. If a well-tested and organized body of design solutions is unavailable, a bottom-up approach may be suitable as a temporary measure until the discipline matures.

Contextual facts: Additional support for a bottom-up approach may be inferred from Shaw's assertion that an expert in a field must know about 50.000 chunks of information (Shaw 1990, with reference to work by Herbert Simon on experts and expert systems). Although a relevant store of facts would seem to comprise also facts about the application context, eg. e-Government, this view of experts as masters of facts seems to emphasize component detail rather than composite system structure.

Technical complexity: In the last decade there has been increasing interest in software architecture as a field of research as well as teaching, see for example (Denning and Dargen, 1994) and (Garlan et al, 1992). Software architecture, which is of interest in an e-government curriculum, entails many difficult technical issues and may lend itself to a bottom-up approach. Indeed, the Software Design and Architecture course mentioned in the survey of the Software Engineering report had several technical prerequisite courses. The recent work on teaching software architecture appears to support this view. The course on Software Architecture developed by Garlan, Shaw, and others that was evaluated in (Garlan et al, 1992) appears to require extensive prerequisite competencies in programming and operating systems, for example

in order for students to comprehend topics such as composition of software entities by pipes or events, and to fulfill the requirements that students design and implement their own prototype system. Indeed it would appear that in-depth knowledge about software architecture presupposes technical insight into the software entities as well as methods for composing them.

Generic design

The generic design approach is to teach design methods that are self-contained and independent of specific context.

There is an element of generic design in all four curriculum reports: although the design part of most knowledge area-definitions in (Denning et al., 1989) seems to consists mostly of illustrative artifacts, there are knowledge areas for which generic methods are listed. This justifies labeling (Denning et al., 1989), and be implication CC 2001, as following a generic approach as a supplement to the main bottom-up approach.

A generic approach is central in a new civil engineering program at the Technical University of Denmark, termed Design and Innovation. The generic focus is evident eg. in titles of courses in the program, such as "Visualization", "Description of technical systems", "Problem solving", "From need to task", etc. (M. M. Andreasen et al., 2002). The program focuses on physical artifacts, rather than software artifacts, but remains of interest as a novel approach to engineering, physical or software-oriented.

Design focus: Generic approaches allow students to focus more on proper design issues than does the bottom-up approach, insofar as students are relieved of the technical details of subsystems. Peter Denning in (Denning 2004) argues that study programs should emphasize principles, including design principles. Denning says that "Computing professionals follow principles of design", and provides a list of examples including abstraction, separate compilation, version control, layering, and several others. These principles are said to be "driven" by five generic design concerns, apparently sitting at the highest level of abstraction: simplicity, performance, resilience, evolvability, and security. There is some resemblance with the literature on project management and the definition therein of a project's major - and possibly conflicting - parameters, including cost, quality, feature set, etc.

Availability: The generic approach presupposes that there is a body of relevant and useful generic methods, of course. This is a matter of dispute, one extreme being that any information system method is undue, as argued in (Baskerville et al., 1994). Methods listed in (Denning 1989) include formal specification and verification and software life cycle models. The Software Engineering curriculum in SE 2004 mentions many generic methods including object-, function-, data-structure-, and aspect-oriented design. In addition to formal methods and methods developed in the context of classes of programming languages, there are of course methods of a less structured and more user-centered nature, including participatory design methods, see eg. (Bødker et al., 2004).

Increasing maturity: New generic methods may emerge as the field of Computer Science matures, as predicted in (Shaw, 1990). Steps towards maturity may involve such relatively simple matters as introducing consistent terminology. In their analysis of the software architecture literature and 'folklore', (Garlan et al, 1992) identified many interesting examples of inconsistent terminology for architectural entities, eg. components, objects, and segments.

Actually, software architecture when taught as sketched by Garlan et al. may be seen to have a strong generic element, in the sense that what subcomponents actually do is largely abstracted away, consistently with the philosophy of module encapsulation. That is, only generic technical properties of subsystems are considered, such as the programming languages used for their implementation, while specific functionality of subsystems is abstracted away.

Top-down design

A top-down approach is for students to begin at the level of the over-all system and then focus on subsystems on an as-needed basis.

Project work: For the purpose of this discussion, the approach is identified with self-managed project work, where students choose problem and problem-solving strategy under instructor supervision. Project work may be seen as top-down insofar as students choose to work with high-level design problems that cannot be solved by straightforward application of knowledge already acquired, thus necessitating subsystem analysis as a means to solve the over-all problem. Informally, a top-down approach can be viewed as the (building) architect's approach, while bottom-up is the approach of the traditional engineer.

Project work plays a central role two Danish Universities formed in the 1970s, Roskilde University and Aalborg University, including in their programs for Computer Science and related disciplines. Projects are also central at the Design and Innovation program at DTU. Background material including (Andreasen, 2002) on the program contains many interesting views on the importance of project work in engineering, including a focus on students' acquiring of competencies in synthesis and evaluation, eg. of trade-offs and alternative designs.

Approach	***Philosophy***	***Rationale***
Bottom-up	Students must understand subsystems before methods for their composition.	Understand language of underlying abstract machine ('physicalism'). Understand design retrospectively, via design results (Denning, 1989). Unavailability of standard methods, due to immaturity (Shaw, 1990). Fact gathering to build context (Shaw, 1990). Technical complexity of subsystems (Garlan et al, 1992).
Generic	There are sound design methods that can be understood at a generic level, independent of subsystems.	Focus on design enabled when time is not unduly spent on technical details of subcomponents. Availability of relevant methods, including formal, structured and object-oriented, and soft user-oriented. Approach may become increasingly more viable as Computer Science matures.
Top-down	Explore subsystems as needed in self-managed project work	Development of analysis, synthesis, and evaluation skills (cf. Bloom). Experience in identifying and solving routine vs. innovative problems. Resembles professional practice.

Table 3. Summary of bottom-up, generic, and top-down approaches to design teaching.

Capstone projects as defined in CC 2001, SE 2004, and CE 2002 are intended to focus on design and implementation. In CE 2002, capstone projects are referred to as the 'culminating design experience' (CE 2002, p 21). (The Information Systems report does not define larger projects.) Thus there is an element of the top-down approach in these reports as well.

Bloom's taxonomy: The Software Engineering report describes the capstone project as an opportunity for students to apply in practice what they have already learned through course work. Despite this view of project work as application of knowledge already acquired, and the entailed downplay of the opportunity to acquire new insights, the report also motivates the project work by an interesting reference to Bloom's taxonomy of levels of cognitive skills. The CE 2002 report characterizes project work as one opportunity (though not the only) for students to achieve the highest level of competence. Bloom's taxonomy of student's depth of knowledge is summarized in Table 4. The three most advanced levels - analysis, synthesis, and evaluation - are strikingly central in design work, whether in professional practice or self-managed student projects.

Professional practice and the routine/innovative task distinction: Work in industry and government resembles project work rather than course work. Moreover, Shaw conjectures in (Shaw 1990) that in disciplines where solutions to standard problems are not synthesized and distributed by a supporting science (due to immaturity, see the subsection on bottom-up above), such insights will be generated through the practical work of professionals, and distributed informally as 'folklore'. Therefore, qua its resemblance with real-world practice, project work may serve as a means of acquiring these crucial, professional competencies, at least to some degree. Relevant competencies that may be acquired include the ability to search for and apply established solutions to routine problems, and to distinguish that sort of activity from the more creative activity of solving the complementary problem type, those design tasks that require innovative work.

Conclusion

Although the ACM and IEEE curriculum reports surveyed in this paper emphasize design in various general remarks, and most notably the Computer Science report discusses in a visionary manner various approaches to the introductory and intermediate levels of the curriculum, eg. programming-first vs. breadth-first, the reports only vaguely indicate how the concrete curricula

Level	***Description***
1: Knowledge	Fact recall with no real understanding behind the meaning of the fact
2: Comprehension	The ability to grasp the meaning of the material
3: Application	The ability to use learned material in new and concrete situations
4: Analysis	The ability to break a complex problem into parts
5: Synthesis	The ability to put parts together to create a unique new entity
6: Evaluation	The ability to judge the value of the material for a given purpose

Table 4. A summary of Bloom's taxonomy of acquired cognitive skills based on (Howard, 1996).

may actually help in building the student's design competencies.

The curricula recommended by the four reports overlap to a striking degree, for example, they all prioritize operating system, databases, and programming-in-the small. The report on Computer Science (CC 2001) tends to identify Computer Science with computing, the term also used in all reports to denote the broader field whose parts are covered by the four disciplines: Not only do the initials CC abbreviate Computing Curriculum; Computer Science is the only discipline defined as a science; and the definition of Computer Science as integrating science and engineering (see Figure 1) comprises, at least, the discipline of Software Engineering. If graduates within all four disciplines acquire essentially the same competencies, albeit with different emphasis, the existence of four independent disciplines may not be justified.

As a basis for discussing alternatives to the 'physicalist' approach of the ACM and IEEE reports, a distinction between approaches of a bottom-up, generic, and top-down nature has been tentatively introduced. The discussion of this taxonomy supports three observations: First, there are indeed many advantages of a bottom-up approach; one key advantage is that technical insight is a solid basis for studying, for example, software architecture design. Second, as Software Engineering and Information Systems mature, an increasing body of relevant generic methods may be available that can be taught and understood before deeper technical competencies are acquired. And third, self-managed project work may be cultivated as a top-down approach to design, and may potentially support the development of crucial and design-relevant competencies such as analysis, synthesis, and evaluation.

References

M. Alonson, E. J. Finn. *Fundamental University Physics. Vol 1.* Addison-Wesley, 1980, 2/e.

K. V. Andersen, N. Jørgensen. *Workshop om digital forvaltning og universitetsuddannelserne*. URL: `http://www.ruc.dk/~nielsj/digital-forvaltning/workshop.html`

M. M. Andreasen, P. Boelskifte, O. Broberg, C. Clausen, C.T. Hansen, L. Hein, M.S. Jørgensen, U. Jørgensen, T. Lenau, T.C. McAloone. *Design-ing.* DTU, 2002. URL: `http://www.designing.dk/"Indstilling om design og innovation.pdf"`

R. Baskerville, J. Travis, D. Truex. Systems Without Method: The Impact of New Technologies on Information Systems Development Projects. *In K. Kendall, K. Lyytinen, J. DeGross. (eds.). IFIP Transactions A8, The Impact of Computer Supported Technologies on Information Systems Development.* Amsterdam: North-Holland, 1992, pp. 241-269.

F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering", *Information Processing* 86, pp 1,069-1,076.

F. P. Brooks. *The mythical man-month. Essays on Software Engineering, Anniversary Edition.* Addison-Wesley, 1995.

K. Bødker, F. Kensing, J. Simonsen. *Participatory IT Design - Designing for Business and Workplace Realities*. MIT Press, 2004.

P. J. Denning. Great Principles in Computing. *Technical Symposium on Computer Science Education (SIGCSE)*, 2004 (Invited Talk), pp 336-341.

P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and Paul R. Young.

Computing as a Discipline. *Communications of the ACM*, Vol. 32 (2), January 1989, pp 9-23.

P. J. Denning and P. A. Dargan. A Discipline of Software Architecture. *Interactions*, Vol 1 (1), pp 55-65, 1994.

The Danish Finance Ministry. E-government. May 2001. Only available in Danish. URL: `http://www.fm.dk/db/filarkiv/6121/Digitalforvaltning.pdf`

D. Garlan, M. Shaw, C. Okasaki, C. M. Scott, and R. F. Swonger. Experience with a Course on Architectures for Software Systems. *Proceedings of the SEI Conference on Software Engineering Education, 1992.* Lecture Notes In Computer Science (Vol. 640).

J. T. Gorgone et al. IS 2002. *Model Curriculum and Guidelines for Undergraduate Programs in Information Systems.* ACM, AIS, and AITP, 2002. URL: `http://www.aisnet.org/Curriculum/IS2002-12-31.pdf`

Å. Grönlund (ed). *Electronic Government: design, applications & management.* Idea Group, 2002.

H. A. Howard, C. A. Carver, W. D. Lane. Felder's learning styles, Bloom's taxonomy, and the Kolb learning cycle: Tying it all together in the CS2 course. *Technical Symposium on Computer Science Education (SIGCSE)*, 1996, pp 227-231.

Joint Task Force on Computing Curricula. *Computer Science*. (CC 2001). ACM and IEEE, 2001. URL: `http://www.computer.org/education/cc2001/cc2001.pdf`

Joint Task Force on Computing Curricula. *Software Engineering 2004*. (SE 2004). ACM and IEEE, 2004. URL: `http://www.computer.org/education/cc2001/SE2004Volume.pdf`

Joint Task Force on Computer Engineering Curricula. *Computer Engineering 2002*. (CE 2002). ACM and IEEE, 2004. URL: `http://www.eng.auburn.edu/ece/CCCE/CCCE-FinalReport-2004Dec12.pdf`

P. Naur and B. Randell (eds). *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, October 1968. Brussels, 1969.

Mary Shaw. Prospects for an Engineering Discipline of Software. IEEE Software, November 1990, pp 15-24.

Mary Shaw. We can teach software better. *Computing Research News*, 4 (4), pp 2-12, September 1992.

Frederik Voetmann Christiansen. Undervisningspraksis og -kulturer i de videregående naturfaglige uddannelser. URL: `http://www.fremtidensnaturfagligeuddannelser.u-net.dk/notater/notat6c.htm`