6

Action-Centered Design

Peter Denning and Pamela Dargan

... the standard engineering design process produces a fundamental blindness to the domains of action in which the customers of software systems live and work. The connection between measurable aspects of the software and the satisfaction of those customers is, at best, tenuous. We propose a broader interpretation of design that is based on observing the repetitive actions of people in a domain and connecting those action-processes to supportive software technologies. We call this activity action-centered design and we propose it as the basis of a discipline of software architecture.

Peter Denning and Pamela Dargan

Peter Denning has had a distinguished career in the computer-science profession. In addition to holding several prominent teaching and research management posts, he has been the president of the Association for Computing Machinery (ACM), the head of the ACM Publications Board, and Editor-in-Chief of the *Communications*, ACM's most widely read publication. He has spearheaded national computing curriculum task forces, including one that produced a milestone document on principles for revising the computer-science curriculum, known widely as the "Denning report."[†]

From this background, we might expect his perspective to be close to the conventional wisdom of computer science, and his advice about design to be tied closely to the methods and slogans of software engineering. But the chapter included here reflects a different direction---one that brings to the center people and their practices, rather than the technologies of computing.

With Pamela Dargan of the Mitre Corporation, Denning set out to find out how successful software designers had managed to create software that users found usable and well suited to their needs. Denning and Dargan did not simply study the systems; they asked the systems' designers and developers what they had done to achieve a good design. As Denning and Dargan report, the answers were surprisingly consistent and surprisingly far from the conventional wisdom on software engineering.

[†] Denning, P. J., D. Comer, D. Gries, M. Mulder, A. Tucker, J. Turner, and P. Young. 1989. "Computing as a discipline". ACM *Communications*, 32 (January), 9-23.

6. Action-Centered Design

Peter Denning and Pamela Dargan

The software landscape is a mixed field of successes and failures. Along with notably successful software packages for every task from payroll to party invitations, we find notable failures, some of them spectacular. As one measure, a 1979 US Government Accounting Office review of nine software projects for the Department of Defense showed that about 2 percent of the allocated funds were spent on software that was delivered and in use, about 25 percent were spent on software that was never delivered, and about 50 percent were spent on software that was delivered, but was never used (Neumann, 1995). This example may represent an extreme case, but everyone in the software industry knows that such problems are widespread and are of large magnitude. Software engineering---a discipline of designed invented in the 1960s to address the "software crisis"---has unwittingly created an illusion that a rigorous process of transforming requirements into systems is the key to reliable design. With this illusion comes a false sense of security that the solution to the crisis is at hand---and that we will obtain it faster by throwing more research collars at the problem.

We believe that the shortcoming is not due to a lack of effort or intelligence among software practitioners. We believe that the problem is of a different kind -- that the standard engineering design process produces a fundamental blindness to the domains of action in which the customers of software systems live and work. The connection between measurable aspects of the software and the satisfaction of those customers is, at best, tenuous. We propose a broader interpretation of design that is based on observing the repetitive actions of people in a domain and connecting those action-processes to supportive software technologies. We call this activity action-centered design and we propose it as the basis of a discipline of software architecture.

6.1 Approaches to Software Design

English dictionaries list no less than 10 different senses of the verb *design*. The primary meaning is "to make or conceive a plan." Software design is concerned with the form and function of a software system and with the structure of the process that produces it. Two principal approaches are practiced. *Software engineering*, which dates to the mid 1960s, is based in the engineering tradition, where design is seen as a formal process of defining specifications and deriving a system from them. *Human-centered design* is much more recent, dating to the late 1980s; it says that designers should immerse themselves in the everyday routines and concerns of their customers. These approaches have complementary strengths and weaknesses. We believe that the two can be brought together into a new discipline, *software architecture*. The central practice of software architecture, *action-centered design*, produces maps that serve as "blueprints" uniting system-oriented engineers and customer-oriented designers.

Let us begin with software engineering. Software engineers believe that most of their work lies in the process that generates a system having the form and function specified by the customer. They refer to the process as the *software lifecycle model*. The most common varieties of the software lifecycle model are the *waterfall model* and the *spiral model*. Vivid descriptions of how this process is organized can be found in the works of Andriole-Freeman (1993), Boehm (1976), DeGrace and Stahl (1990), and Dijkstra (1989).

Unlike other engineers, however, software engineers have not achieved success in the engineering design process. They have not developed a systematic method of producing software that is easy to use, reliable, and dependable: they have not developed a discipline of software production.

Many explanations have been proposed for this anomaly. The most popular ones posit the notion that software is complex and violates the continuity laws that engineers are accustomed to -- changing a single bit in a program can produce drastic changes in behavior. Software is seen as a "radical novelty" for which our customary engineering analogies are misleading (see Dijkstra, 1989, and Parnas, 1985). However comforting they might be, none of these explanations has produced a practical means for systematically producing usable, dependable software.

The engineering design process operates from three assumptions:

1. The result of the design is a product (artifact, machine, or system).

2. The product is derived from specifications given by the customer; in principle, with enough knowledge and computing power, this derivation could be mechanized.

3. Once the customer and designer have agreed on the specifications, there is little need for contact between them until delivery.

The software crisis is usually seen as a breakdown in the application of this methodology, especially due to faulty input in the form of incomplete or incorrect specifications.

The human-centered design school emerged in Europe in reaction to the shortcomings of what its adherents call "product-centered design" (Floyd, 1992). These designers say that the software crisis is in fact a failure of customer satisfaction (Denning, 1992a). They say that the standard engineering design process offers little to connect the actions of designers with the concerns of users; and that the interaction between these people is limited to the requirements and specifications documents and to the final sign-off. Donald Norman (1993) argues that the product-centered design process focuses primarily on the machine and its efficiency, expecting humans to adapt. In contrast, he says, the human-centered design process leaves to humans the actions that humans do well -- empathizing, perceiving, understanding, solving problems, listening for concerns, fulfilling commitments, satisfying others, serving interests -- and leaves to machines what humans do not do well, such as performing repetitive actions without error, searching large data sets, and carrying out accurate calculations.

Human-centered design of computer systems attempts to understand the domain of work or play in which people are engaged and in which they interact with computers, and then to design the computers and software to facilitate action. Anthropologists play a significant role in this work. Human-centered design operates from three assumptions:

1. The result of the design is a satisfied customer.

2. The process of design is a collaboration between designers and customers; it evolves and adapts to their changing concerns, and it produces a specification as an important byproduct.

3. The customer and designer are in constant communication during the entire process.

6.2 A New Interpretation of Design

A discipline of software design must train its practitioners to be skilled observers of the domain of action in which a particular community of people engage, so that the designers can produce software that assists people in performing those actions more effectively. The phrase *domain of action* is meant to be broad; it includes specialized domains, such as medicine, law, banking, and sports; and general domains such as work, career, entertainment, education, finance, law, family, health, world, dignity, and spirituality. Software engineers must reframe design from a process of deriving a software system from specifications to *a process of supporting standard practices in a domain, through software that enables more effective action*.

We recommend this reframing because the assessment of whether software is useful, reliable, and dependable is made by the people who act in a domain. By focusing on a system and its specifications, the traditional engineering process loses sight of those people, of their common concerns and of their actions. The process cannot offer a grounded assessment of quality, because many of the factors influencing quality are not observable in the software itself (Denning, 1992a). Human-centered design, at least, does not lose sign of the community. As presently constituted, however, human-centered design lacks formalisms and is not capable of making connections systematically between user concerns and the structure of software.

Many software engineers today are using object-oriented design (OOD) as a way to frame the requirements-analysis stage of the engineering design process, and are using object-oriented programming (OOP) as an approach to the implementation. Having observed this, we asked a number of prominent software engineers this question: “What is the concern that makes people find object-oriented programming so attractive?” Their answers were instructive. Most saw object-oriented programming as another fad, rather than as the “silver bullet” that will end the software crisis. Most thought that object-oriented programming appeals to some intuitive sense about how the software will be used, and thereby reduces the apparently complexity of the software.

We then asked the designers of several award-winning software packages: “What does it mean for a design to be intuitive and judged as complexity reducing?” The packages were:

- Quicken (by Intuit), a personal financial accounting system (see Chapter 13).
- MeetingMaker (by On Technologies) and Synchronize (by Crosswind Technologies), systems for scheduling meetings and managing calendars in groups.
- Topic (by Verity), a system for retrieving documents from a database based on their text content.
- Macintosh user interface (by Apple) (see the profile in Chapter 4).

We asked these designers what they had done to achieve a good design. There was a surprising level of unanimity in their answers:

- Pick a domain in which many people are involved and that is a constant source of breakdowns for them. (For example, the designer of Quicken chose personal finance.)
- Study the nature of the actions people take in that domain, especially of repetitive actions. About what do they complain most? What new actions would they like to perform next? (In personal finance, the repetive actions include writing checks and balancing the checkbook; the most common complaints include errors in arithmetic and discrepancies between bank statements and personal balance sheets; and the most-sought new action is to generate reports automatically for income-tax purposes.)

- Define software routines that imitate familiar patterns of action. Users will have little to learn to get started, because the software merely helps them "do the obvious." Include functions that permit actions most users have wished they could do, but could not do manually. (In personal finance, these functions include persenting screen images of template checks, allowing electronic payments, and providing a database that records transactions by income-tax type.)

- Deploy prototypes early in selected customer domains. Observe how people react and what kinds of breakdowns they experience. Because the customers frequently shift concerns, especially after they are seasoned users, it is necessary to provide means of observing shifting concerns and of taking these shifts into account in the next version of the design. Thus, there are beta-test sites, individual follow-up sessions, hot lines, highly attentive technical and customer support staff, suggestion boxes, bug advisories, and the like. It is of central importance to stay in communication with customers.

All the designers said that they did not pay much attention to standard software-engineering methodology. Several said that the internal structure of their code is ugly and is not well modularized. When fixing bugs they made patches; when the system got too patchy, they declared the next version and redesigned it completely.

One way to summarize these findings is this: Software systems have customers. Quality means customer satisfaction. Customers are more likely to be satisfied by software that is transparent in their domain of work, because it allows them to perform familiar actions without distraction, and to perform new actions about which previously they could only speculate. Customer satisfaction is not static: Customers change expectations, and the software must evolve to track their shifting expectations.

We also reviewed the published papers of people who had designed software systems that were give the ACM Best System award. We found a similar set of concerns expressed here, even though the initial users of these systems were typically technical specialists (see Denning and Dargan, 1994).

6.3 Pattern Mapping as a Basis for a Discipline of Design

A discipline of software design should be capable of training its practitioners to fulfill systematically promises to build and install software systems that are judged useful and dependable by their customers. The key to transforming software design and engineering into a customer-centered discipline is the development of a method of mapping human actions to software functions in a way that is intelligible to clients, designers, and engineers simultaneously.

The field of architecture is rich in useful analogies and practices. Architect Christopher Alexander (1979) says that the job of an architect is to give expression to the patterns in the use of space that permit the building's occupants to carry out their daily actions effectively. He says that surprisingly few patterns are needed to describe and generate buildings -- a dozen or so suffice for a typical home and three or four dozen for a typical office -- and that the immense variety of buildings arises from the infinite number of ways these basic patterns can be combined. He says that these patterns are not objects, such as bricks, beams, boards, floors, or ceilings, but are relationships among simpler patterns and the environment. When everyone involved knows the pattern language of the building, he says, it is easy for the builders to construct an edifice that is harmonious with the lives and work of the building's users.

The architect's blueprint is a map (actually, an interrelated set of maps) that expresses the patterns and their relationships and provides a common language among architect, builder, and client. Coplien and Schmidt (1995) have proposed a set of patterns for software construction. Even so, the fields of software design and engineering have no generally-accepted method of mapping analogous to the blueprint, no agreement on the basic patterns of human action that will be composed in a software system. Although we do not yet have a formal method to construct such maps, we do know what we would want from them. The maps would provide ways

1. To convey the patterns of action of the domain in which the software will be used, in terms of its basic distinctions, repetitive processes, standards of assessment, strategies, tools, breakdowns, and driving concerns

2. To connect the linguistic structure of the domain to the software structures that will support the patterns, and to guide software engineers in implementing those structures

3. To provide a basis for measuring the effectiveness of the implemented system in practice

The domain actors would find the map a useful depiction of how they work (step 1); the software producers would find it useful to configure client-server networks, databases, and applications to support the actors' work (step 2); and observers would use it to guide measurements (step 3).

These maps go beyond the standard elements of computer code or program specifications. They map the domain of action, rather than just the system being built. The basic patterns that these maps should cover are

- A set of *linguistic distinctions* (verbs, nouns, jargon, etc.), around which people in the domain organize their actions. (In personal finance, these include checks, ledgers, banks, bank accounts, bank statements, merchants, bills, fund transfers, deposits, credits, interest, and the like.)

- A set of *speech acts* by which domain participants declare and report states of affairs, initiate actions, signify completions, and coordinate with others. (In personal finance, these include “pay”, “deposit”, “withdraw”, “transfer funds”, “reconciliation successful”, “prepare tax report”, “cancel payment”, and the like.)

- A set of *standard practices* (recurrent actions, organizational processes, roles, standards of assessment) in which members of the domain engage. (In personal finance, these include paying monthly bills, reconciling the ledger, putting money into savings, preparing quarterly tax summaries, maintaining positive balances in accounts, earning good interest, having a minimum liquidity level, having a maximum debt level, getting a credit rating, and the like.)

- A set of ready-to-hand *tools and equipment* that people use to perform actions; a tool is ready-to-hand if the person using it does so with skill and without conscious thought. (In personal finance, these include pens, calculators, checkbooks, databases, tax forms, monthly reports, and the like.)

- A set of *breakdowns*, which are interruptions of standard practices and progress caused by tools breaking, people failing to complete agreements, external circumstances, etc. (In personal finance, these include errors writing or coding checks, missing payments, discrepancies between ledger and bank statement, lost deposits, errors in credit reports, lost checks, inability to compile tax data, unresponsive customer service departments, broken modems, and the like.)

- A set of *ongoing concerns* of the people in the domain -- common missions, interests, and fears. (In personal finance these include a working banking system, good credit rating, liquidity, low debt, steady income, accurate tax data, good return on investment, fear of tax audit, and the like.)

This overall framework is sometimes called the *ontology* of the domain (see Winograd and Flores, 1987). Put simply, an ontology is a conceptual framework for interpreting the world in terms of recurrent actions. Building an ontology of the domain in which software will be used, representing it as a pattern language in a standard notation, and coordinating the work of builders, is the central activity of a software architect. This skill is analogous to the architect’s skill in creating sketches and blueprints, in using them to coordinate builders, and later using them to help clients assess the results. We call this *action-centered design*.

An important aspect of action-centered design is its emphasis on speech acts, on use of language, and on repetitive actions. This perspective is a distinct departure from the traditional functional analysis of a domain, which describes the domain as a *process-network* -- a network of interconnected input-output functions that operate on physical or information objects. Speech acts such as "I request", "I promise", "I have completed the task", and "I am satisfied" are important because they are the motivating force for action. Without them, no task would be declared, initiated, or completed, and no one would know whether anyone else was satisfied. (So few people regularly use the second two of these acts that miscoordination and lack of trust and satisfaction are all too common in the workplace (Denning, 1995).)

Human beings engage in repetitive processes in which they coordinate action by standard speech acts; processes that become mindless routines are ripe for automation. The skilled designer knows how to observe the repetitive processes by watching what people say to one another (verbally, in writing, electronically, body language, or through other communications media), and then to offer tools that allow people to complete their repetitive processes faster or more reliably.

The notation of business process maps (see the Profile on Business Process) appears to be well suited as a starting point for the maps of repetitive coordination processes (Medina-Mora et al, 1992, Denning, 1992b). This notation needs to be extended to show how a process-network is triggered by people acting in their coordination processes. The software system would implement the clients, servers, networks, operating systems, and applications needed to perform the tasks and transfer data and signals among them; it would also detect people's speech acts and use them to trigger functions in the software system.

We propose an interpretation of design that is

- Focused primarily on satisfying the customer rather than on satisfying the system's specifications

- Grounded in a language-action perspective, rather than in a system dataflow network perspective

- Based on observations of concerns, breakdowns, standard practices, institutions, and recurring actions, and on production of means to connect those observations with software structures.

Action-centered design consists of observing the ontology of a domain, then constructing a workflow map, a task-network map, and the connections between them. The maps can be used by the software architect to review with the client how the system will satisfy each concern, and to coordinate the implementation of the system with the software engineers. The rudiments of this process can be seen in the examples of the successful software packages and systems that we studied, and in the statements of their designers.

Note that we are *not* advocating that anyone abandon process-network diagrams and other formalisms of software engineering. We *are* advocating that designers learn to look much more broadly, rigorously observing the domain of action and then coupling software tools with actions that people perform and assessments that people make.

We propose this interpretation not as a final answer, but as a preliminary step -- an opening for a new direction in software design.

6. References (all referred to in text)

Alexander, C. 1979. *The Timeless Way of Building*. Oxford.

Andriole, S. J., and P. A. Freeman. 1993. "Software systems engineering: the case for a new discipline." *IEEE Software Engineering Journal* (May), 165-179.

Boehm, B. 1976. "Software engineering." *IEEE Trans. Computers*, C-25 (December), 1226-1244.

Coplien, J., and D. Schmidt. 1995. *Pattern Languages of Program Design*. Addison-Wesley.

DeGrace, P., and L. Hulet-Stahl. 1990. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Yourdon/Prentice-Hall.

Denning, P. J. 1992a. "What is software quality?" *ACM Communications*, 35 (January).

Denning, P. J. 1992b. "Work is a closed loop process." *American Scientist* (July), 314-317.

Denning, P. J., and P. A. Dargan. 1994. "A discipline of software architecture." *ACM Interactions*, 1 (January), 55-65.

Denning, P. J., and R. Medina-Mora. 1995. "Completing the loops." *ORSA/TIMS Interfaces*, 25 (May), 42-57.

Dijkstra, E. W. 1989. "On the cruelty of really teaching computer science." *ACM Communications*, 32 (December), 1398-1404.

Floyd, C., H. Züllighoven, R. Budde, and R. Keil-Slawik (eds.) 1992. *Software Development and Reality Construction*. Berlin: Springer-Verlag.

Medina-Mora, R., T. Winograd, R. Flores, F. Flores. 1993. "The ActionWorkflow Approach to Workflow Management Technology." In *The Information Society*, 9, 4 (October-December), 391.

Neumann, P. 1995. *Computer-Related Risks*. Addison-Wesley.

Norman, D. 1993. *Things That Make Us Smart: Defending Human Attributes in the Machine Age*. Addison-Wesley.

Parnas, D. 1985. "Software aspects of strategic defense systems." *ACM Communications*, 28 (December), 1326-1335.

Winograd, T., and F. Flores. 1987. *Understanding Computers and Cognition*. Addison-Wesley.