DOI:10.1145/2535915

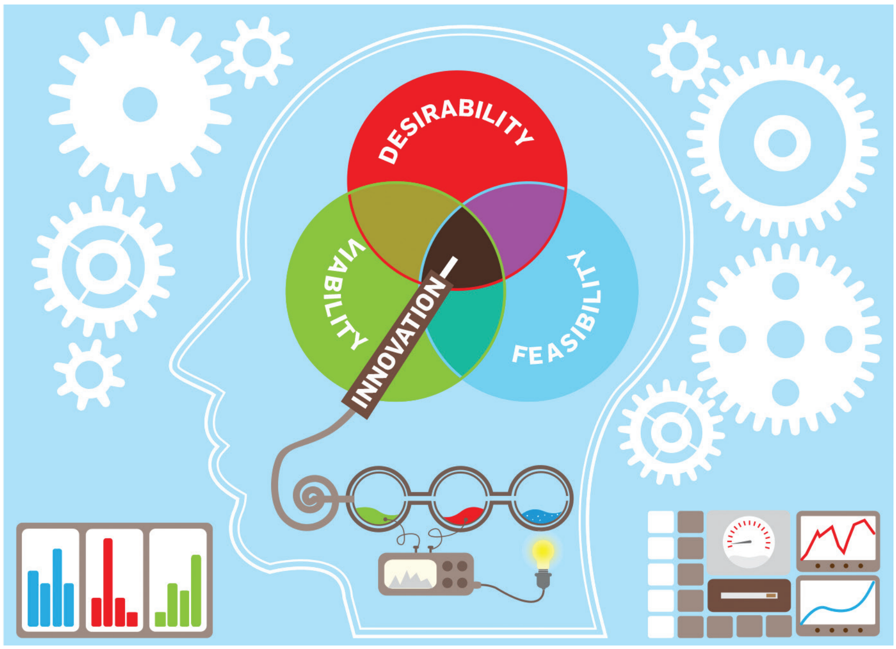
Peter J. Denning

The Profession of IT
Design Thinking

Design thinking is the newest fashion for finding better solutions to problems. Combining it with computational thinking offers some real possibilities for improving software design.

HAVE YOU NOTICED design thinking? In recent months a spate of news reports, *60 Minutes* segments, and TV news reports have told how the IDEO Company and its founder, David Kelley, have developed an innovative new approach to product design. Many social service and government organizations are now looking at IDEO's design thinking as a path to process innovation in their organizations. Practitioners trying to solve wicked problems are also finding fruit in design thinking. I would like to share some reflections on these reports.

I will review computing's long history of involvement in design. Some of our software development communities, especially those under the headings of agile, participatory, and user-centered, have been design thinkers for a long time. I will offer a note of caution about the claim of some media advocates that design thinking will speed up the processes of innovation.



Design

Design is familiar in many fields including fashion, products, architecture, engineering, science, and software development. Design is a process where we create and shape artifacts that solve problems. In software, for example, design means crafting software that does jobs users want done. Software designers intentionally support practices, worlds, and identities of the software's users. Designers have accumulated much practical wisdom that is expressed with design principles such as separation of concerns, modularity, abstraction, layering, wholeness, utility, resiliency, beauty, and timelessness. Design principles in computing guide us to ways of building machines whose behaviors are useful and meaningful in their user communities.

Seasoned designers constantly run experiments with prototypes to learn how well the machines might work and how users might react to them. Maurice Wilkes stressed this point in his 1967 ACM Turing Lecture,[5] saying that a great strength in the early days was the willingness of research groups to construct experimental computers without necessarily intending them to be prototypes for commercial production. Their experiments produced a body of knowledge about what would work and what would not work. In his 1995 memoir, Wilkes strongly criticized the more recent trend to ignore the historical development and try to design from scratch, as in the personal computer world. Without the knowledge of what worked and what did not, designers have tended to repeat the same mistakes.[6] Like Fred Brooks (author of *No Silver Bullet*), Wilkes believed that good design is a skill set with many dimensions, well worth cultivating.

Since its beginnings in the 1940s, software has had a reputation of being extremely error prone. Programmers have always been frustrated by the amount of time they need to spend locating mistakes in their own programs, and in protecting software and data from external errors. This has not been easy given the vulnerabilities inherent in the chain of transformations that designers must master (see the accompanying figure):

1. The specification does not accurately represent the designer's intention, or is based on misunderstandings of user expectations.

2. The programmer makes mistakes, for example, by introducing bugs or approximations that cause the software to violate its specifications. Moreover, the compiler might contain bugs or Trojan horses that cause the machine code to be not equivalent to the source program.

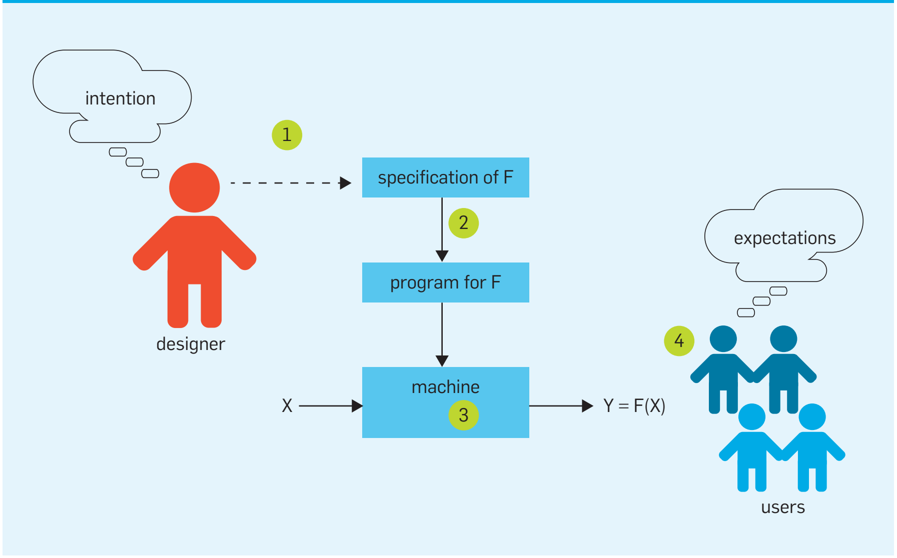
3. The machine itself contains bugs, defects, malfunctions, intrusions by other buggy machines or attackers, and other factors that cause it to misbehave while executing its basic operations.

4. The users' expectations of what the machine's job is differs from what they see the machine doing.

When combined with the general concern for getting work done on time, these error sources have led to five traditional criteria for software design:

- *Requirements—Does it have a clear purpose?* The designer knows what job the machine is intended to perform and can state the requirements precisely as a specification. Articulating requirements is a challenge because interviewing the intended users about what they want is notoriously unreliable.
- *Correctness—Does it work properly?* The behavior of a source code program provably meets precise specifications. Correctness is challenging because requirements are often fuzzy and proofs are often computationally infeasible. Experimental methods are often the only practical way to learn user requirements, arrive at precise specifications, avoid intractability, and test prototype behavior.
- *Fault tolerance—Does it keep working?* The software and its host systems can continue to function despite small errors, and will refuse to function in case of a large error. Redundancy supports fault tolerance by duplicating hardware and data so that a failed component can be bypassed by other still-working components and datasets. Error confinement supports fault tolerance by structuring the operating environment so that no process has access to any object other than those it needs for its computation and by limiting (or eliminating) the super user state.
- *Timeliness—Does it complete its work in time to be useful?* The system completes its tasks within the expected deadlines. Supporting techniques include algorithm analysis, queueing network analysis, and real-time system deadline analysis.
- *Fitness—Does it align well with the user environment?* Fitness is challenging because assessments like dependability, reliability, usability, safety, and security are context sensitive and much of the context is not obvious even to the experienced designer. The famous architect Christopher Alexander advocated an approach to design that was immersed in the context of how dwellers used buildings.[1] Alexander's work inspired a group of software designers to found a "software pattern community" devoted the ideals of good software design. A software pattern characterizes a large number of situations that a programmer is likely to encounter, and offers guidance on how to structure the program to best fit the pattern. The pattern community appeals to my sense of empiricism because they are relentless about testing ideas with potential users and learning from the feedback. A good introductory book on the topic is *Pattern Languages of Software Design* (Addison-Wesley, 1995), by James Coplien and Douglas Schmidt. Coplien is one of the founders of the pattern community.

Four sources of errors in computations.



Software and Computer System Design

Despite their best efforts with tools, languages, and project management, software experts felt they could not keep up with the demands for ever-larger reliable systems. In 1968, they founded the field of software engineering to apply standard engineering methods to meet two concerns:

1. Increasing reliability and managing risk by eliminating or confining errors.

2. Seeing software projects through to completion (delivery) and subsequent evolution (maintenance).

Within computing various schools of thought have developed around specific approaches to these concerns. These schools have advanced "process models" like waterfall or spiral, or "design approaches" such as participatory, user centered, agile, or pattern. They are all after the same thing, but they weigh the criteria in different ways. Barry Boehm argued that the standard engineering design approach of careful, almost rigid process was at the strict end of a planning spectrum and agile methods were at the flexible end.[2] He thought that careful planning is needed when reliability, safety, and security are important and that agility is needed when usability and evolvability are important. He exhorted the careful planning and the agile schools to collaborate on finding a common ground for better systems.

Design Thinking

Design thinking means to intentionally focus the design around the concerns, interests, and values of the users. The current strand of design thinking that has captured much public attention and interest originated with industrial product design in the company IDEO founded in 1991 by David Kelley in partnership with several other design firms.

In 2006, Kelley founded the Stanford Design Center, which has become an intellectual center for a design thinking movement. The IDEO philosophy emphasizes that design is a team sport with three principal values:

1. Many eyes—Design teams include diversified expertise such as engineering, human factors, communication, graphics, ethnography, sociology, and more. Each team member's unique perspective helps the other members see things they would not ordinarily see.
2. Customer viewpoint—Design teams visit customer places to interview them and watch what they actually do, including their reactions in extreme "stress" cases.
3. Tangibility—Design teams build prototypes and mockups, try them out, and learn from the feedback and reactions.

There is a big interest in the public sector to apply design thinking to its own problems; the standard government approach of one-size-fits-all does not work for the diverse communities agencies are trying to serve. IDEO has helped agencies improve their processes, often with excellent results that receive a lot of publicity.

Design thinking has also been helpful in addressing wicked problems.[4] Design teams generate moods of collaboration, often leading to breakthroughs for resolving their wicked problems. When the design team brings together all the various stakeholders of a company, they are often able to win the commitments from multiple divisions of the company to see new ideas through to production. It should be noted that design thinking is not the only successful method for generating collaboration: so also do Appreciative Inquiry, Charrettes, and the Straus Method.

Design thinking is already deeply embedded into the software pattern community.

Design and Innovation

I am wary of the claim that design thinking speeds up innovation. Design and innovation are related, but not the same. Innovation is concerned with getting a community of people to adopt a new practice often organized around an artifact or process created by a designer. The only time design thinking might accelerate innovation is when the ideation stage has been blocked by lack of ideas or by conflicts among key stakeholders.

Master designer Don Norman puts this point differently.[3] He says designers working alone have a low rate of technology adoption. Technological entrepreneurs pursue the business opportunities and work hard, often over many years, to win the commitments for adoption. Design and entrepreneurship are both needed to transform new proposals into adopted practices.

Disruptive innovations, which reconfigure entire communities, are quite rare. When they do happen, they seem spectacular. Well-designed artifacts are often cited as the causes of the innovation. But this impression seldom holds up under scrutiny. Consider the iPhone. The iPhone is partly a story of design (Steve Jobs had help from IDEO with the artifact) but it is mostly a story of entrepreneurship: Apple miniaturized components, created portable apps, put many sensors into the phone, generated a community of software app developers, cloned the Apps Store from the successful iTunes store, partnered with AT&T and later Verizon, created a new class of data plans for phones, and built a new operating system—iOS—to support it all. Apple caused the innovation. The iPhone was the tip of an iceberg of practices and business arrangements that made it work.

Seasoned innovators work with a "90% rule"—90% of the work in achieving an innovation goes into the adoption phase. Ideation, where design thinking produces its value, is the other 10%. But many media reports would have you believe that design thinking gets you 90% of the way to innovation. I take issue with these pundits. While a collaborative team can get things done faster, often the design team is not a collaborator with the production, marketing, PR, and community outreach divisions of a company.

Putting It All Together

Design thinking calls attention to creativity and imagination in the ideation process, emphasizing collaborative, diverse, customer sensitive design teams. It also emphasizes frequent customer feedback from prototypes that elicit their reactions. Design thinking is already deeply embedded into the software pattern community, which is part of agile software development. That community has accumulated a large set of insights into what makes for successful software design. You need look no farther than that community to see how to put design thinking to work in software development.

Computing's traditional view of design is strongly flavored by its concern for building artifacts that are error tolerant or error free. Design thinking is strongly flavored by its concern for understanding what job an artifact does for its users. If the two kinds of thinking were blended together, some significant advances in software design and development would surely follow. C

References

1. Alexander, C. *The Timeless Way of Building.* Oxford University Press, 1979.
2. Boehm, B. Get ready for agile methods, with care. *IEEE Computer* (Jan. 2002), 64–69.
3. Norman, D. Technology first, needs last: The research-product gulf. *ACM interactions 17*, 2 (Mar.+Apr. 2010).
4. Roberts, N. Wicked problems and network approaches to resolution. *The Int. Public Mgmt. Review 1*, 1 (2000).
5. Wilkes, M. Computers then and now. (1967 Turing Award lecture). *JACM 15*, 1 (Jan. 1968), 1–7.
6. Wilkes, M. *Computing Perspectives.* Morgan Kaufman, San Francisco, CA, 1995.

Peter J. Denning (pjd@nps.edu) is Distinguished Professor of Computer Science and Director of the Cebrowski Institute for information innovation at the Naval Postgraduate School in Monterey, CA, is Editor of ACM *Ubiquity*, and is a past president of ACM. The author's views expressed here are not necessarily those of his employer or the U.S. federal government.