Alan Cooper and the Goal Directed Design Process

By Hugh Dubberly

Originally published in *Gain*
AIGA Journal of Design for the Network Economy
Volume 1, Number 2, 2001

Dubberly Design Office
2501 Harrison Street, #7
San Francisco, CA 94110

415 648 9799

Alan Cooper is not your typical graphic designer—he's an engineer *and* a card-carrying member of the AIGA. He inhabits both worlds and has something important to say to designers and other engineers.

Cooper is not one to say things softly. He's outgoing, quick to offer an opinion or an aphorism, and seems to like nothing better than a healthy debate. His favorite topic: what's wrong with the software that increasingly fills our lives.

Cooper has been designing software since the arrival of personal computers more than 25 years ago. There are few people who have thought as long and deeply about what good software design is and about how to produce it. Much of that thinking both comes from and infuses his work at Cooper Interaction Design, the 70-person firm he founded and runs in Palo Alto, California.

Cooper is surprisingly generous, even zealous, about sharing what he has learned. He lays out his beliefs in two books: *About Face: The Essentials of User Interface Design* [IDG Books,1995] and *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity* [SAMS,1999]. In *Inmates*, Cooper provides a detailed argument on the need for change. In sum, his argument is this:

Computer chips are increasingly powerful, making computer power less and less expensive. As a result computers are being built into more and more products. And where there are computers, there must also be software. And where there is software, very often, there is user interaction. Already, it's difficult to find new cars, appliances, or consumer electronics that do not require users to interact with software.

So what's the problem?

It is this: software does not reveal itself through external form—something mechanical devices tend to do. And in software, the cost of adding one more new feature is almost nothing, whereas adding features to mechanical devices almost always increases their cost. Cooper argues that software is thus less constrained by negative feedback acting to limit complexity than mechanical devices have been. The result is pure Rube Goldberg: software with feature piled upon feature. The trouble is that each incremental feature makes a product *more* difficult to use. That leaves us with products that are increasingly hard to use—and with growing frustration as we try to use them.

In the traditional software development process, lots of people inside a company—and many times customers as well—ask for features. In many companies, the resulting list of features often becomes the *de facto* product plan. Programmers make this approach worse by picking or negotiating their way through the list, often trading time for features. In such a process, Cooper points out, it is difficult to know when a product is complete.

The heart of the problem, he concludes, is that the people responsible for developing software products don't know precisely what constitutes a good product. It follows that they also do not know what processes lead to a good product. In short, they are operating by trial and error, with outcomes like customer satisfaction achieved by little more than blind luck.

Cooper believes things don't have to be so bad and points to the fact that the industry is young and still learning how to make software. He sees an analogy in the language of film, a process of telling interesting stories with movies that was not inherent in the invention of the movie camera. After the appearance of cameras and projection devices, the art and craft of filmmaking also had to be invented. Cooper believes we're near a similar point of invention in the process of developing software. (The parallels between movie making and software development are striking: computer visionary Ted Nelson has gone so far as to suggest that software development is a branch of movie making).

Cooper advocates five significant changes to the conventional methods of software development in his goal-directed design process:

1. Design first; program second.

Old way: programming began as soon as possible—applying design at the end if at all. Or, in more progressive environments, programming and design happened concurrently.

"The single most important process change we can make," Cooper says, "is to design our interactive products completely before any programming begins."

See Diagram: Evolution of the Software Development Process.

2. Separate responsibility for design from responsibility for programming.

Old way: programmers made significant decisions about how users interact with the software—often while in the middle of programming.

Allowing the same person to design and program creates a conflict of interest. Programmers want the product to be easy to code while designers desire to make the product easy to use.

3. Hold designers responsible for product quality and user satisfaction.

Old way: management held programmers responsible for product quality—since they're the ones who made it.

This point has an important corollary: The flip side of taking responsibility for product quality is receiving authority to decide how the product behaves and what it looks like. That means management has to be clear with programmers that the design spec is not merely a suggestion but rather a plan they must follow. Cooper says, "The design team must have responsibility for everything that comes in contact with the user. This includes all hardware as well as software. Collateral software such as install programs and supporting products must be considered, too."

Cooper's next point, the heart of his approach, is a new take on an old idea: focus on the customer.

4. Define one specific user for your product; then invent a persona—give that user a name and an environment and derive his or her goals.

Old way: managers and programmers talked about "the end user" without being specific—allowing the term "user" to stretch to fit the situation.

A persona is a composite portrait of an idealized user: a single sheet of paper with name, picture, job description, goals, and often a quote. Cooper notes, "We print out copies of the cast of characters and distribute it at every meeting. . . Until the user is precisely defined, the programmer can always imagine that he is the user."

Goals derived from the persona are the focus of Cooper's entire process. (See Diagram: Evolution of the Software Development Process).

User goals inform or direct all design decisions. "Personas are the single most powerful design tool that we use. They are the foundation for all subsequent goal-directed design. Personas allow us to see the scope and nature of the design problem . . . [They] are the bright light under which we do surgery."

Cooper's approach differs from task-analysis-based approaches by focusing first on goals to ensure that the right tasks are identified. "Goals are not the same thing as tasks. A goal is an end condition, whereas a task is an intermediate process needed to achieve the goal…. The goal is a steady thing. The tasks are transient," he says.

Finally, Cooper suggests a new way of organizing the design team.

5. Work in teams of two: designer and design communicator

Old ways: one programmer, or one interaction designer, or one interaction designer and one visual designer.

Assign two people to all project teams: a designer to be responsible for the product concept and a design communicator (very like a writer) to be responsible for the description of the product. (See Diagram: Designer/Design Communicator). This pairing resembles the art director and copywriter pairing common in advertising, although Cooper is insistent in pointing out that the role of the design communicator goes beyond just writing and documentation.

Where do these changes lead?

Cooper maintains that goal-directed design will lead to software products that are more powerful and more pleasurable to use. He outlines five major benefits:

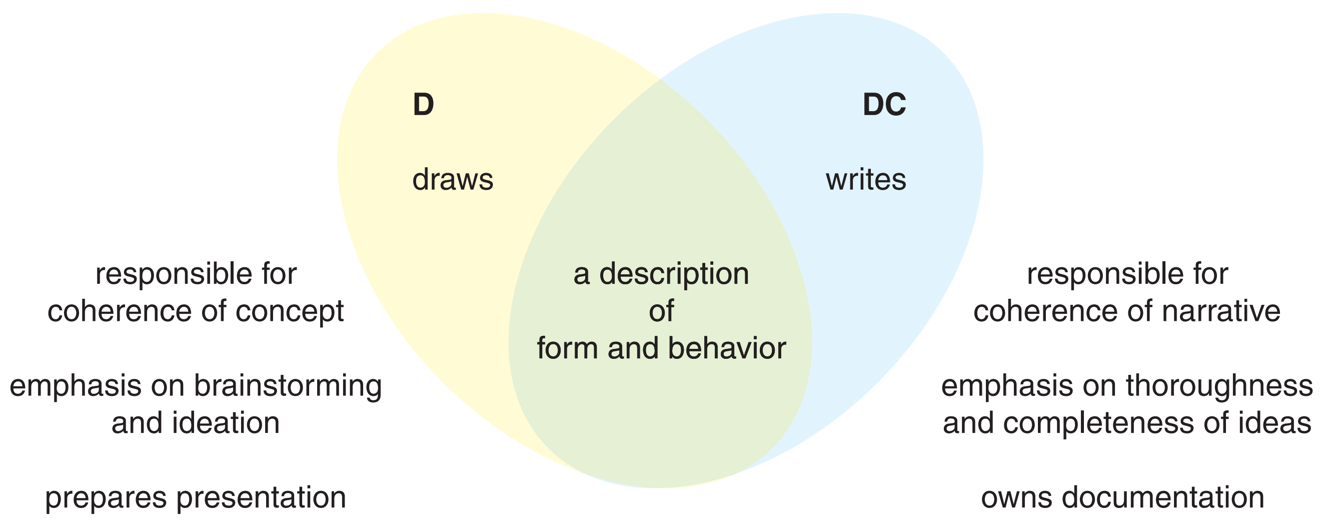
1) Improved product quality
2) Reduced development time—which leads to reduced cost
3) Improved documentation (Reducing the complexity of the software reduces the time spent explaining software problems and frees up time to explain how the software can really help users).

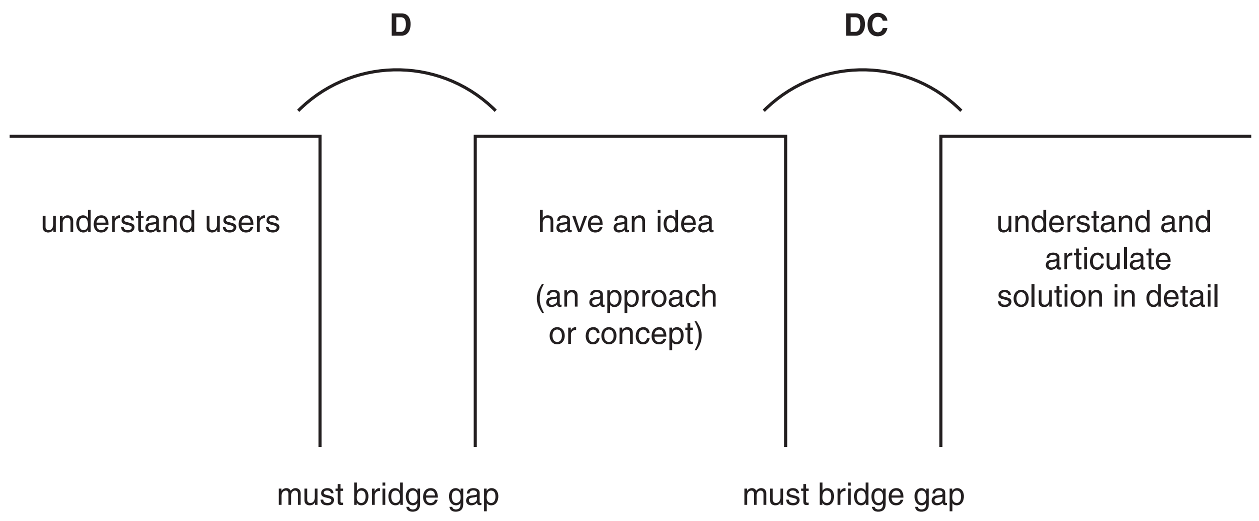
Designer/Design Communicator The Power of Two-person Teams

Cooper organizes projects around two-person teams. One is a designer, the other, a design communicator. This method contrasts with the common pairing of an interaction designer and a visual designer—a method that may diminish the visual designer's role.

Rather, Cooper's approach is a true team, as in ad agencies where it's not always clear which team member wrote the headline or which came up with the concept. In the end, shared work is both more fun and also higher quality.

at Cooper	**Designer - D**	**Design Communicator - DC**
on the X-files	Agent Mulder	Agent Sculley
with a musical	Composer (Rodgers)	Lyricist (Hammerstein)
in a Platonic dialogue	Citizen (knower of truth)	Socrates (gadfly, questioner)
in an ad agency	Art director (Lee Clow)	Copy writer (Steve Hayden)
to fly – you need both	Kite	String


D
draws
DC
writes
responsible for
coherence of concept
emphasis on brainstorming
and ideation
prepares presentation
a description
of
form and behavior
responsible for
coherence of narrative
emphasis on thoroughness
and completeness of ideas
owns documentation


D
DC
understand users
have an idea
(an approach
or concept)
understand and
articulate
solution in detail
must bridge gap
must bridge gap

Originally, programmers did it all:

In the early days of the PC software industry, smart programmers dreamed up useful software, wrote it, and even tested it on their own. As their businesses grew, the businesses and the programs became more complicated.

Managers brought order:

Inevitably, professional managers were brought in. Good product managers understand the market and competitors. They define software products by creating requirements documents. Often, however, requirements are little more than a list of features, and managers find themselves having to give up features in order to meet schedules.

Testing became a separate step:

As the industry has matured, testing has become a separate discipline and a separate step in the process. Today, it's common to find 1 tester for every 3 or 4 programmers. This change illustrates that the programmer's role is not fixed but still evolving.

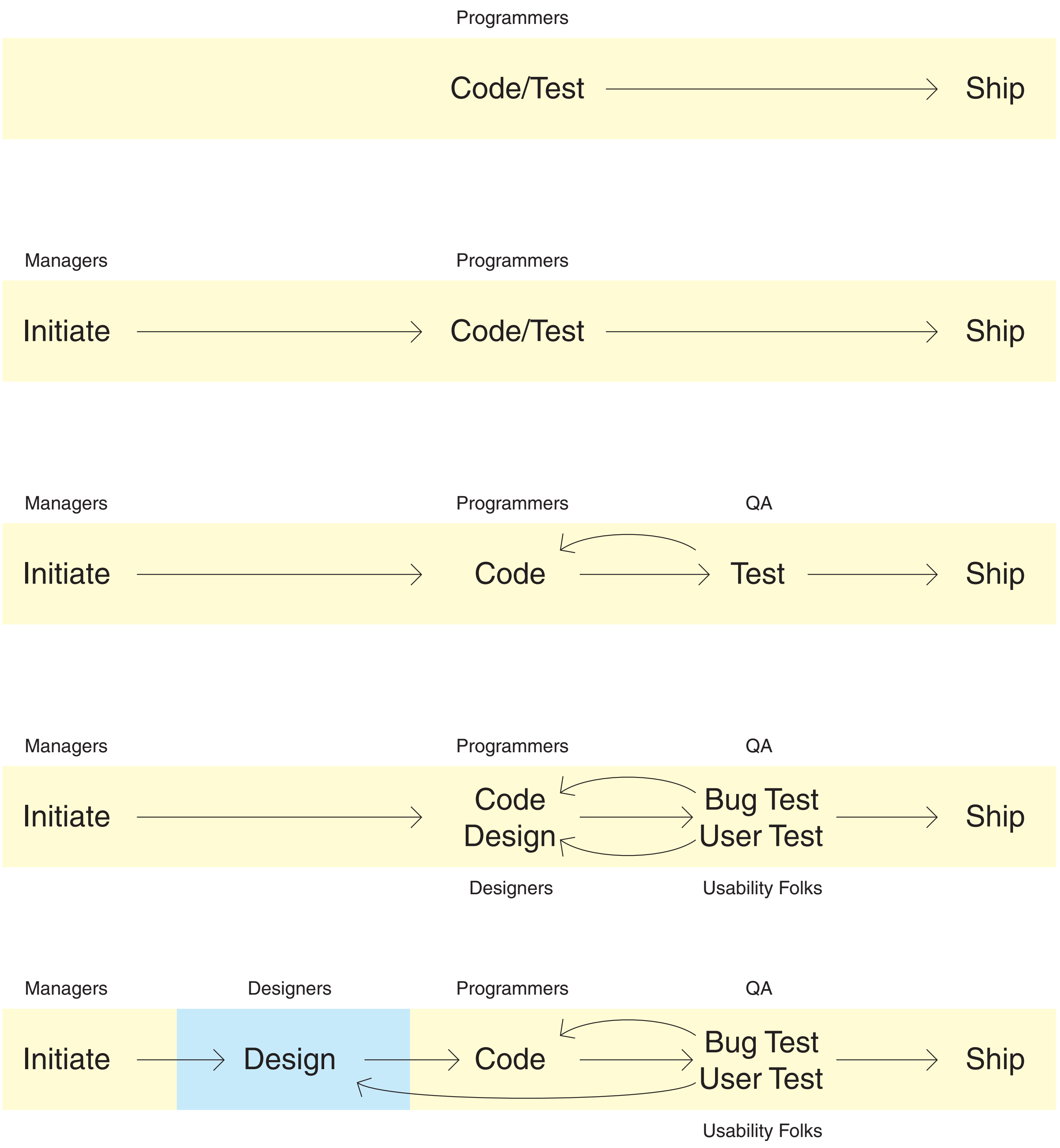
Today, common practice is to code and design simultaneously:

In the move from command-line to graphical user interface, designers became involved in the process—though often only at the end. Today, common practice is for simultaneous coding and design followed by bug and user testing and then revision.

Cooper insists that design precede programming:

In Cooper's goal-directed approach to software development, all decisions proceed from a formal definition of the user and his or her goals. Definition of the user and user goals is the responsibility of the designer—thus design precedes programming.

Evolution of the Software Development Process



Four Dimensions of Design

Cooper focuses on an area of design that traditional designers do not often explore: the design of behavior. Yet, all design affects behavior: Architecture is about how people use spaces as much as it's about form and light. And what would be the point of a poster if no one acted on the information it presented?

One way of making sense of the difference in focus between Cooper's work and more traditional design is through the lens of history. In the first half of the twentieth century, designers focused primarily on form. Later, designers became increasingly concerned with meaning, for example, product designers and architects introduced vernacular and retro forms in the 1970s; the trend continues today with retro-styled automobiles such as the PT Cruiser.

Within the last five years, a growing group of designers have begun to talk about behavior—the experience users have with a product. (Of course some designers such as Aaron Marcus and John Rheinfrank, have focused on behavior for a long time).

These concerns—form, meaning, and behavior—are not exclusive. Great work combines them—as Maya Lin did, for example, in the Vietnam Veterans Memorial which is at once a carefully considered form, a series of layers of meaning, and a profoundly moving experience.

Form, meaning, and behavior are not a closed set; already we see hints of a fourth order of design (Richard Buchanan's phrase)—the design of possibility, opportunity, co-creation, or collaborative systems. Again, it is not something new; merely a new area of focus.

The Making of an Interaction Designer

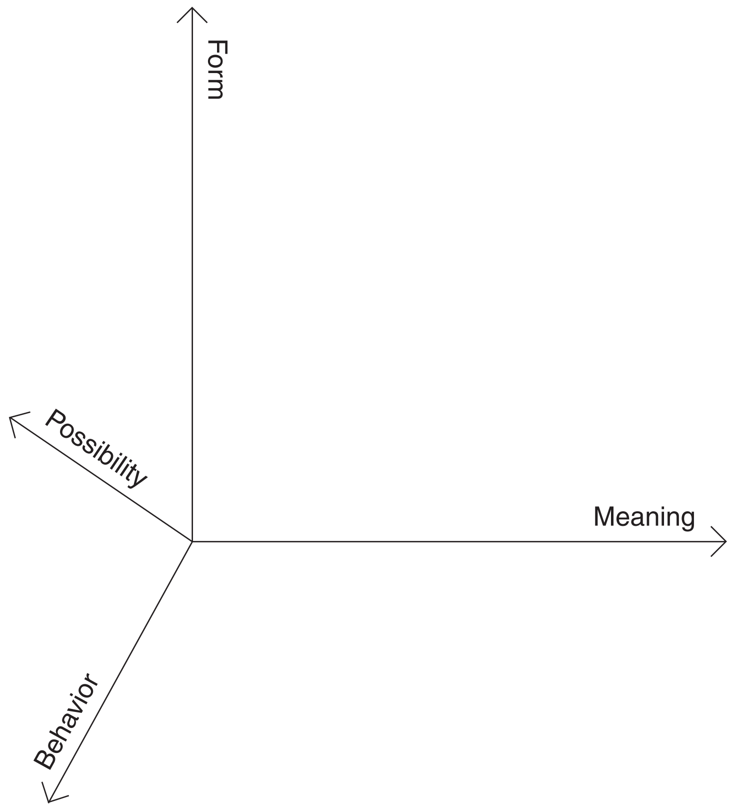
Alan Cooper's fascination with computers was first triggered by the flashing lights of an IBM System 360 that he saw while visiting a Zurich bank in 1972. After that encounter, he enrolled in data processing classes and learned to program.

But Cooper's interest in design predates his interest in computers. "One morning when I was 14," he recalls, "I woke up with a bolt of crystal clarity and knew that I wanted to be an architect . . . I read every book in my high school library on architecture." Architecture, urban planning, and transportation design remain passions, and Cooper often describes software design in terms of architecture and vice-versa, "The architect translates the needs of the user into terms that could be understood by the builder," he says.

Cooper applied to study architecture at UC Berkeley's College of Environmental Design, but despite winning a full Regent's Scholarship, he never attended. Instead, after Cooper saw a magazine ad for the Altair, an early personal computer, he put off college in order to start a software company, just as Microsoft founders Paul Allen and Bill Gates did. That was in 1975, before there was a PC industry and before there was a software industry.

Cooper borrowed $10,000 from his father (who took out a second mortgage on the family house to provide the money) and started a company with his high school friend, Keith Parsons. Structured Systems Group (SSG) developed and sold turn-key accounting systems, offering both a personal computer and the software to run it at prices far below comparable minicomputer-based systems of the day. They soon realized that they didn't need to sell the computers and began to sell software independently, a new idea at the time. SSG also began publishing Gordon Eubanks' CBASIC, an early programming language. In their book, *Fire in the Valley: The Making of the Personal Computer*, Paul Freiberger and Michael Swaine describe SSG as "one of the first companies to deliver business software for microcomputer . . ." and a progenitor of the "general software company," in its time on a par with Microsoft and Digital Research.

SSG grew to 25 people but after four years Cooper left. He formed a new company, Access Software. While at SSG Cooper had been the chief programmer doing much of the coding as well as designing the software. At Access, Cooper's role was chief designer. "If the user came in contact with it, I defined it." Instead of doing the programming



himself, he hired others to implement his vision of the interface and left them free to organize the code as they thought best. After two years with Access, Cooper joined his friend Gordon Eubanks at Digital Research, taking a role focused on design. He stayed little more than a year. Frustrated with the development process and priorities at Digital Research, Cooper left to work on his own doing what he calls "speculative product development."

Cooper worked on several projects including a visual programming language. It enabled programmers to build applications quickly and easily by clicking on file names and dragging them into a structure. Cooper showed his program to Bill Gates who proceeded to buy it, replace Cooper's programming language with BASIC (a new version of what had been Microsoft's first product), and eventually publish the hybrid as Visual Basic. Visual Basic was wildly successful because it made easy what until then had been difficult. Windows had previously required programmers to know C, a demanding programming language. As Cooper explains, "Visual Basic let's you code without learning 600 Windows SDK [Software Developer Kit] calls." Gates showed his gratitude by bestowing Microsoft's Windows Pioneer Award on Cooper. Cooper notes that Gates also gave him "a one-line resume: Father of Visual Basic."

While doing his speculative development work, Cooper toyed with the idea consulting. There were lots of opportunities to program, but he did not want to code other people's software designs. Instead, he wanted to design software products, but he didn't think anyone would pay him merely for designing. Finally, in 1992, after speaking on an industry panel, he took a gamble and announced that he was henceforth working as a software design consultant. Two people on the panel offered him work.

In 1994, Cooper Interaction Design was busy enough to take on two employees. Seven years later, it has 70 employees with a range of backgrounds: technical writing, software project management, tech support, graphic design, the humanities, physics, architecture, computer science, and industrial design. They occupy offices in two two-story buildings located a block apart on the edge of the Stanford campus and Stanford research park in Palo Alto—deep in the heart of a Silicon Valley that desperately needs to put the user first.

Who Provides What to Whom in the Software Development Process

🟦 Bottom-up output 🟨 Top-down input

		Senior Management receive	Product (Project) Management receive	Software Designers receive	Software Programmers receive	Software QA receive	End user receive
Senior Management	provide	-	compensation stable environment vision of company authority for product goals resources	compensation stable environment vision of company	compensation stable environment vision of company	compensation stable environment vision of company	a company that can deliver products vision of company
Product (Project) Management	provide	a product that will be profitable, delivered on time and budget	-	vision of product requirements doc authority for visible product behavior goals resources arbitration	vision of product requirements doc authority for code (invisible behavior) goals resources arbitration	vision of product requirements doc authority for release goals resources arbitration	vision of product a finished product
Software Designers	provide	-	time estimates design plan behavior specs finished artwork	-	behavior specs finished artwork answers to questions	behavior specs	a product with a satisfying experience
Software Programmers	provide	-	time estimates engineering plan engineering spec finished code	tech opportunities tech constraints answers to questions feedback (on behavior spec)	-	engineering spec feedback (on test plan) candidate code bug fixes	a product that's fast, efficient, and meets behavior specs
Software QA	provide	-	time estimates QA plan tested code	feedback (on behavior, though this is rare)	bug list, definition, and prioritization	-	a product with a minimum of defects
End users	provide	payment input feedback	feedback on bugs	input feedback (on behavior)	-	-	-

The Science of Goal-Directed Systems

By Paul Pangaro

Further study of goals might lead to software that adapts to the goals of individual users, learning and responding as it's used. For help in this quest, designers can turn to a branch of science that studies goal-directed activity.

A classic example of goal-directed activity is the steering of a ship toward a destination. The captain aims directly for a point on shore but is driven off-course by wind or tide. Seeing the discrepancy, the captain makes a correction based on the magnitude and direction of the error. Through iteration of this loop—action, feedback, evaluation, re-action—the holder of the goal does his best to reach the goal.

In the 1940s the aptness of this example caused Norbert Wiener and Arturo Rosenblueth to name a new discipline after it: "cybernetics," from the Greek kybernetes or "steersmanship." [Norbert Wiener, *Cybernetics: or Control and Communication in the Animal and the Machine*. John Wiley and Sons, 1948.]

Cybernetics begins with the observer's identification of a "system" that uses feedback to modify actions in pursuit of a goal, regardless of what materials comprise the system. Though early discussions were often about mechanistic systems, practitioners in cybernetics—who came from psychology, anthropology, mathematics, biology, physics, and sociology—immediately understood the power of the "goal-directed" perspective for modeling human activities.

Of course, human beings themselves are "goal-directed systems, and this recognition is an important step toward improving the software design process. Everything that we design should reflect the terminology and dimensions of its user, if that user is to clearly take action, absorb feedback, and evaluate the discrepancy between a current and desired state. Because these processes are clearly iterative, cybernetics would also counsel designers to view *the end-user's activity* as essentially one of prototyping, that is, iteratively converging on higher and higher fidelity versions of some ideal, final goal.

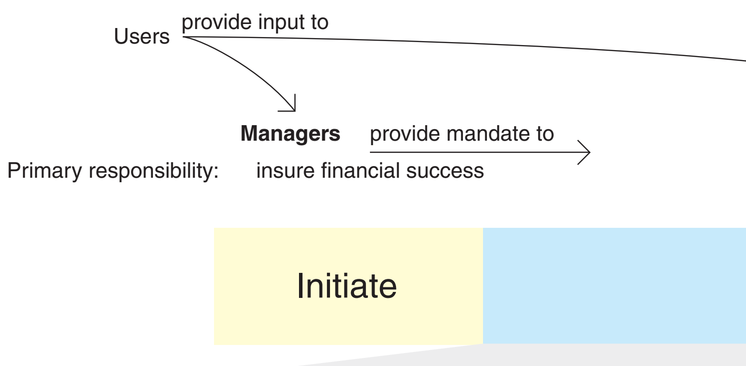
When interacting with human colleagues we must express our goals in order to be understood and to collaborate. Cybernetics suggests that we look at software in a similar way—that we ask how software might hold representations of our goals, help us reflect on them, and even participate in their development.

Cybernetics further suggests that interaction design may come to embrace the end-user as a *designer* of goals, not merely an achiever of them. As software better supports users in achieving goals they have already formulated, designers may find ways to focus more explicitly on helping the end-user who is not yet certain of an end-goal. Interaction design might then bear surprising results—when the end-user can express, evaluate, and modify representations of his or her goals.

Goal Directed Design

Cooper puts user goals at the center of the software design process. That process is part of a series of office practices which depend on the talent and skills of designers and on their application of principles and patterns throughout the process.

This diagram shows the process proceeding in steps from left to right. It leaves out feed-back loops and iteration which are necessary for producing good work.



Research and Analyze

(focus in the first half, continuing throughout)

Opportunities, Constraints, and Context

Who will use the product?
What problem will it solve for them?

Activity:	Define intent and constraints of project	Review what exists (e.g. documents)	Discuss values, issues, expectations	Apply ethnographic research techniques	Define typical users
Result:	Scope	Audit	Interviews	Observations	Personas
	desired outcomes time constraints financial constraints general process milestones (Scope may be loose or tight.)	business plan marketing plan branding strategy market research product plan competitors related technology	management domain experts customers partners sales channel (This step leads to a project mandate.)	use patterns potential users their activities their environments their interactions their objects (tools) (aeiou framework from Rick Robinson, Sapient)	primary secondary supplemental negative served (indirectly) partner customer organizational
Artifact:	Project Brief	Summary Insights	Tapes Transcripts Summary Insights	Tapes Transcripts Summary Insights	Notes
Meetings:	Briefing	-	Interviews	Chalk talk (early findings)	-

lead to →

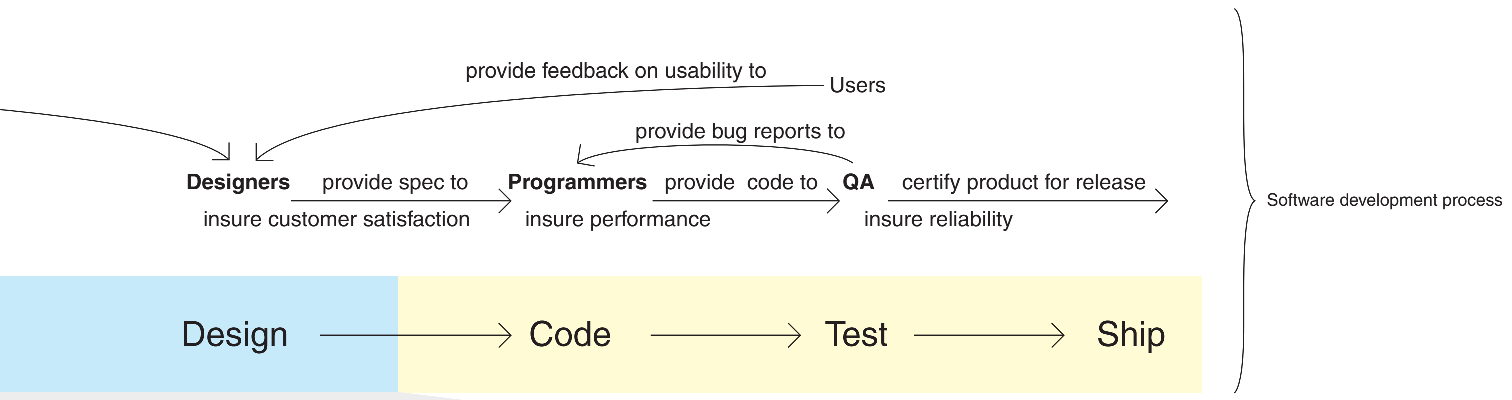
Design Office Practices

The way the office is set up and run – the environment, the spoken and unspoken rules – affect the work. Cooper's staff describes several key practices:
- goal-directed design process
- collaborative environment and common purpose
- D/DC team structure (see separate diagram)
- egoless design
- appropriateness of assignments
- commitment to education
- commitment to enhance process
- assessment and self-assessment

Designer Talent and Skills

A designer's native abilities and background also affect the work. Cooper looks for people with these skills:

- analytic	- empathic
- conceptual	- interpersonal
- visual	- brainstorming
- written	- imagination
- communications	



The goal-directed design process takes place within a larger software development process.

Synthesize and Refine
(ongoing throughout, focus in the second half)

Form, Meaning, and Behavior
What is it?
How will it behave for users?

Goals	Concept	Scenarios	Elements	Framework	Spec
Deduce what users want	Imagine a system to help users reach goals	Tell stories about using the system	Derive components based on users	Organize the components	Refine details; describe models
life end experience personal practical corporate false	problem definition vision definition design imperatives (May require changes in scope.)	day-in-the-life key-path error set-up	information objects functional objects control mechanisms	object relationships conceptual groupings patterns logic / narrative flow navigation structure	appearance language flow / behavior product character product story
Notes	Formal Document Problem Statement Vision Statement	Notes Storyboards	Lists Sketches Diagrams High-level data models	Sketches Flow Diagrams	Formal Document Demonstration Prototype
Chalk talk with management	Presentation	-	-	Chalk talk with programmers	Presentation

Goals $\xrightarrow{\text{drive*}}$ Concept

* spark
inform
motivate
filter
organize
prioritize
inflect
validate

Throughout the goal-directed design process, designers apply other practices, their talent and skills, as well as principles and patterns.

Design Principles

Principles guide the choices designers make as they create. Principles apply at all levels of design from broad concept to small detail. For example:
- Do no harm. (Hippocrates)
- Meet user goals.
- Create the simplest complete solution. (Ockham, Fuller)
- Create viable and feasible systems.

Design Patterns

Design patterns are recurring forms or structures which designers may recognize or apply – during analysis and especially during synthesis. Christopher Alexander, "A Pattern Language," provides examples of patterns for architecture; Cooper collects patterns for software interaction. For example, a common pattern is dividing a window into two panes: the left smaller pane provides tools or context and the right larger one provides a working space or details.

How to Build a Successful Product

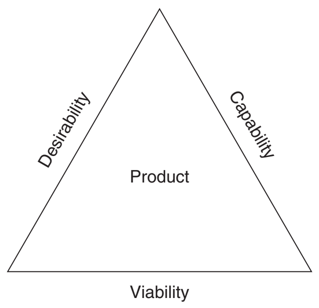
Underlying Cooper's approach to design is the premise that products must balance business and engineering concerns with user concerns.

He begins by asking, "What do people desire?" Then, he asks, "Of the things people desire, what will sustain a business?" And finally, he asks, "Of the things people desire that will also sustain a business, what can we build?" A common trap is to focus primarily on technology while loosing sight of viability and desirability.

Understanding the importance of each dimension is only the beginning. That understanding must be turned into action. We're most familiar with this process along the business dimension: create a business model and then develop a business plan. That process works for technology and users as well. Cooper's goal-directed design process is an analog to the business planning process. It results in a solid user model and a comprehensive user plan.

The user plan determines the probability that customers will adopt a product. The business plan determines the probability that the business can sustain itself up to and through launch—and that sales will actually support growth thereafter. And the technology plan determines the probability that the product can be made to work and actually delivered.

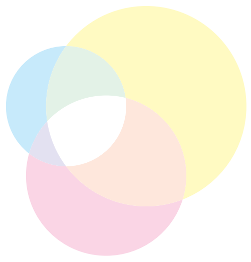
Multiplying these three factors determines the overall probability that a product will be successful.



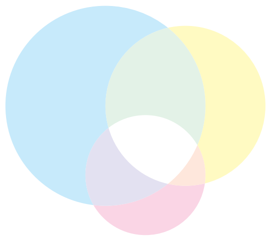
Larry Keeley proposed the original model (above) on which this diagram (far right) builds. Keeley's model described the three primary qualities in a high-technology business.

Others have proposed measures of quality that have three dimensions:
- Vitruvius: solidity, commodity, delight
- ISO 9241: efficiency, effectiveness, satisfaction
- Cooper: hot, simple, deep
- and of course: fast, cheap, good

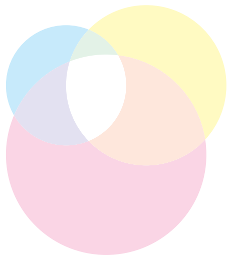
Cooper applies this model to three software giants who have failed to find a balance:



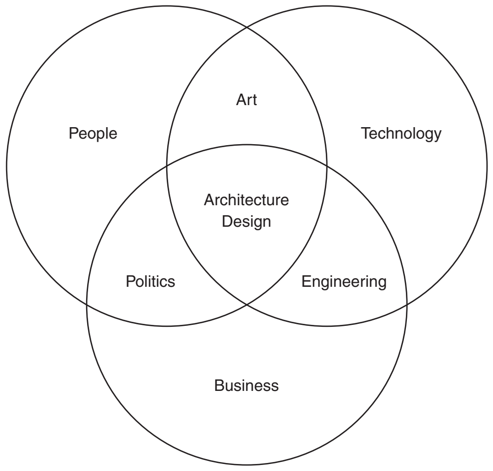
Novell emphasized technology and gave little attention to desirability. This made it vulnerable to competition.



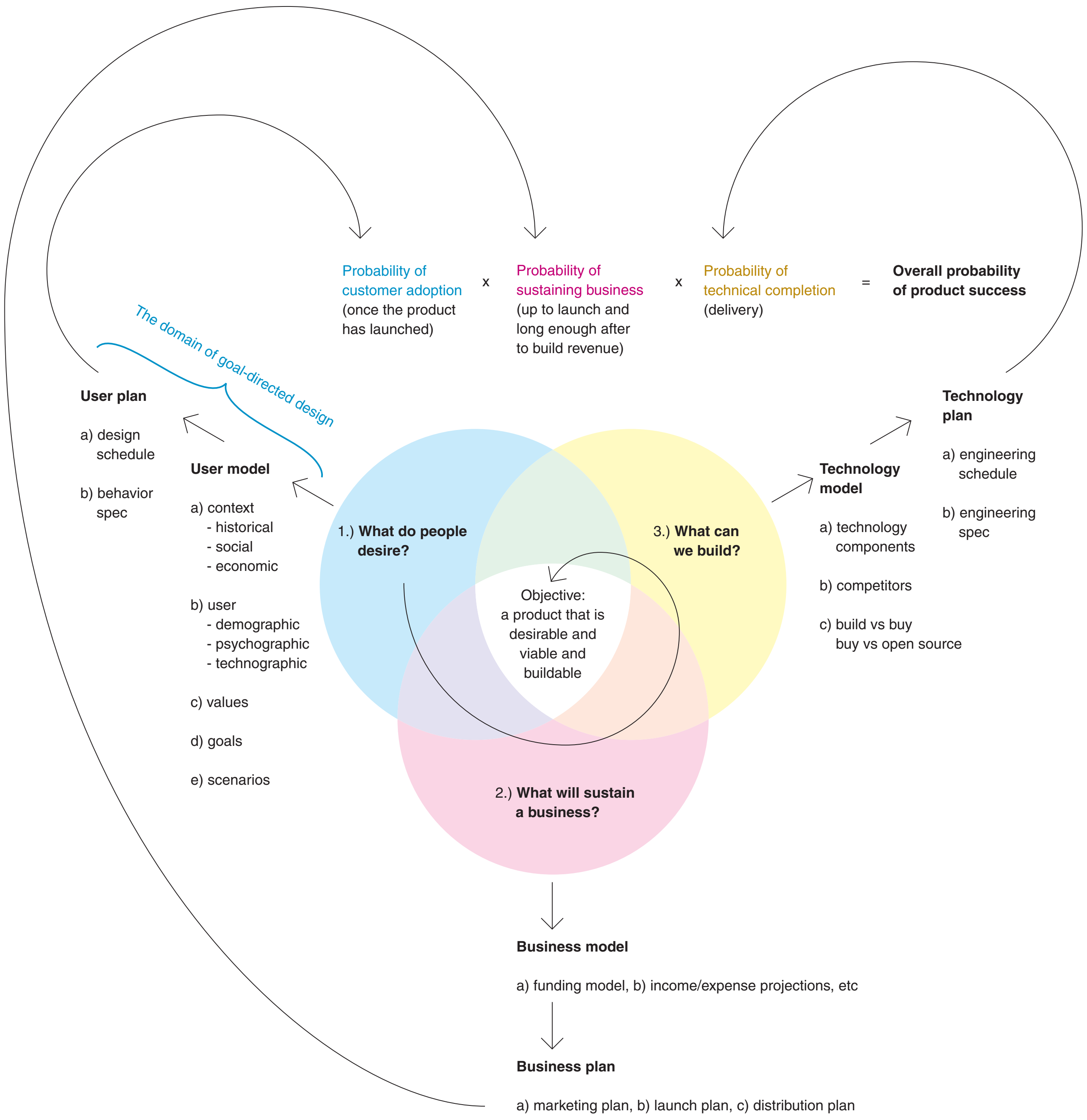
Apple emphasized desirability but has made many business blunders. Never-the-less, it is sustained by the loyalty its attention to users creates.



Microsoft is one of the best run businesses ever, but it has not been able to create highly desirable products. This provides an opening for competitors.



Cooper relables the 'teething rings' people, business, and technology. He places his first love, architecture, at the center – along with software design.


Probability of
customer adoption
(once the product
has launched)
x
Probability of
sustaining business
(up to launch and
long enough after
to build revenue)
x
Probability of
technical completion
(delivery)
=
Overall probability
of product success
The domain of goal-directed design
User plan
a) design
schedule
b) behavior
spec
User model
a) context
- historical
- social
- economic
b) user
- demographic
- psychographic
- technographic
c) values
d) goals
e) scenarios
1.) What do people
desire?
3.) What can
we build?
Objective:
a product that is
desirable and
viable and
buildable
2.) What will sustain
a business?
Technology
model
a) technology
components
b) competitors
c) build vs buy
buy vs open source
Technology
plan
a) engineering
schedule
b) engineering
spec
Business model
a) funding model, b) income/expense projections, etc
Business plan
a) marketing plan, b) launch plan, c) distribution plan