15
The Divorce of Design

Around the sixteenth century, there emerged in most of the European languages the term "design" or its equivalent. . . . Above all, the term indicated that designing was to be separated from doing.

MICHAEL COOLEY [1988]

Wright Brothers' first flight, Nags Head, North Carolina
Library of Congress

The Divorce of Design from Use and from Implementation

One of the most striking 20th-century developments in the design disciplines has been the progressive divorce of the designer from both the implementer and the user.

Consider the 19th- to turn-of-the-20th-century inventors. Edison fabricated working versions of all his inventions in his laboratory. Henry Ford made his own car. Wilbur and Orville Wright built their airplane with their own hands.

A century later, what computer engineer can *make* his own chips, much less start with sand and copper? What airplane designer is a master of the complex manufacturing processes that will build the plane, much less the complex software that will dynamically stabilize it? What architect does his own structural engineering and earthquake strengthening?

Similarly, in many disciplines the designer is divorced from the user as well. In architecture, the designer of a hospital, a crematorium, a nuclear-fuel-processing plant, a biophysics laboratory, brings little personal experience as a user and must elicit expected user behavior from representative users or, worse, user surrogates who are themselves removed a step or two from real users. Few naval architects have commanded a ship, much less wielded one in battle.

This is in sharp contrast with the situation only a generation ago. Today's cars were designed by senior engineers who had spent their teenage years taking old cars apart under the proverbial shade tree. Today's senior communications experts mostly had ham licenses and probably crafted one-tube radios in school. Some of today's British senior mechanical engineers were the product of 1-3-1 "sandwich" programs: a year of hands-on training with the company, three years in university at company expense, another year of hands-on training before starting to design. Many of America's engineers are the product of co-op programs that interspersed college with hands-on industrial experience.

Fortunately, there are exceptions to the divorces. Software engineering, for example, is still so young that system architects were once programmers. The designers of personal products such as the iPod, the iPhone, and cars are, first of all, users, and their own use-vision illuminates their designs.

The designers of UNIX and especially the Open Source designers of Linux start with their own needs, build tools for their own use, and share with their own peers. I reckon this accounts for both the use success and the user passion.

Why the Divorces?

The first reason is obvious. The stunning 20th-century advances in all implementation technologies demand specialization and protract learning times. Keeping up with just earthquake engineering or just manufacturing with composite materials is now a full-time job.

A second reason is less obvious but perhaps as strong. The things we design are so much more complex that just their *design* demands specialization, protracted learning times, and all the designer's energies. There are now few unsophisticated technologies. Consider the complex manufacturing process of the simple Twinkie, where good taste has to be combined with good shelf life and with continued separation of the filling from the cake.[1]

Fallout from the Divorces

So what? What consequences can we see? *Miscommunication abounds.* Architects build elegant buildings that are hard to work in. Engineers design control panels that nuclear reactor operators find confusing. Over-specified implementations cost way more than they should, with little added function or performance. Both the user-designer link and the designer-implementer link narrow radically in bandwidth. Communication *between* people is always much poorer than communication *within* a person. Instances of disastrous, costly, or embarrassing miscommunication abound.

Remedies

The first implication is that designers must recognize that the 20th-century divorces have occurred, and that much extra deliberate and focused effort must be marshaled to mitigate their painful effects.

Remedy 1: Use-Scenario Experience

Even a small amount of use-scenario experience is better than none. Even a good simulation of a use experience is better than none. Full-scale mock-ups enable dry runs of kitchen or cockpit scenarios. So do virtual environments.

When assigned to design an operator's console for the IBM Stretch computer, I had only hearsay evidence as to what operators actually did, much less the relative frequencies and importances of their several tasks.

The Stretch team stopped for two weeks in the summer while most took vacations. So I went to the computation center that operated 709 computers for the Poughkeepsie laboratories, and I applied to be an apprentice operator for two weeks.

It was immensely informative. Mostly, I mounted tapes, but I soaked in the rhythms of a scientific computer center and sharply watched what the chief operators did.[2]

This "user" experience led to the design of the first operator's console to be program-controlled (essentially a close-connected terminal) rather than directly reflecting and affecting the hardware, a capability that enables multiple consoles for multiple operators, and a flexible allocation of tasks among operators, as well as online interactive debugging of programs.

I must admit that the overly fancy console I designed seems to have been rarely used in any Stretch installation in the ways I envisioned. Online interactive debugging did not become a reality until considerably later, partly because Ted Codd's multiprogramming operating system for Stretch was an option, not the standard Stretch software.[3]

The experience became more fruitful when I was engaged in the Operating System/360 design. All the factors were in place for online interactive debugging, and the previous exploration led to a leaner terminal and full software support.

A similar experience was a semester's sabbatical in Dave and Jane Richardson's biochemistry laboratory at Duke. Daily exposure helped me understand their needs for molecular graphics tools for studying protein structure and function.

Philippe Kruchten systematized this sort of exposure when he was the lead architect for Canada's air-traffic control system:

> *All the software people were sent for hands-on training on air-traffic control, going to ATC classes, then spending days sitting next to controllers in a live Area Control Center, trying to understand what was the essence of their activity. Similarly, the ATC specialists were sent to courses such as Object-Oriented Design, Programming in Ada, to reach the point where there was enough common vocabulary for them to efficiently work together and leverage each other's skills.*[4]

Remedy 2: Close Interaction with Users via Incremental Development and Iterative Delivery

Harlan Mills's system of incremental development and iterative delivery is the best way to stay quite close to users right from the very start of the project.[5] One builds a minimal-function version that works; then one gives it to users to use, or at least to test-drive. Even products being built for a mass market can be tested on a sample of users.

In my own practice, building interactive graphics systems as tools for scientists, I have usually been surprised by early user reactions to our prototype systems. Almost invariably I have made wrong assumptions about how they would use the new tool.

My team spent some ten years realizing our dream of a "room-filling protein" virtual image. My idea was that the chemists could more readily find their way around in the complex molecule by knowing where the C-end and the N-end were positioned in the physical room. After many disappointments, we finally had a suitable high-resolution image in a head-mounted display. The chemist could readily walk around in the protein structure to study areas of interest.

Our first user came for her biweekly appointment; all went well and she moved about quite a bit. Next session, same thing.

Third session: "May I have a chair?" A decade's work shot down by one sentence! The navigation assistance wasn't worth the physical labor.

We had a similar experience with a radiation-treatment planning system. The radiologist's task is to find directions of multiple beams that will impinge on the tumor while avoiding sensitive organs such as eyes. We hung the patient's semitransparent virtual body in space, so the physicians could walk around and sight through from all viewpoints. No, they much preferred to sit and rotate the virtual patient through all angles.

Remedy 3: Concurrent Engineering

Designers need to dig more energetically and personally into the actual experiences and processes of implementation. Even an isolated and unrepresentative implementation experience can wonderfully inform a designer's often idealized or inchoate vision of how implementation is done. I recommend it highly.

There is a danger that a modest sample experience of implementation will unduly influence a design, if the designer's personal experience is all that is available—it is by nature unrepresentative. Probably the best balance is achieved with concurrent engineering as the main design practice. Here, the true implementers are intimately involved in the design process; their broad experience provides the balance for a designer's limited implementation examples. (In the software field, this same practice sometimes is called just an agile method.)

Pulling implementers forward into the design process makes its own demands. Shipyard workers who are skilled at following standard engineering drawings may be less skilled at envisioning the finished construct from the standard plans and sections, hence unable to catch mistakes or to foresee implementation "gotchas." Augmenting the standard plans and sections with richer visuals, even virtual-environment explorations, may provide the tools that lubricate the concurrent design process.

Remedy 4: Education of Designers

Design curricula simply must include both techniques for and practice at *understanding* users' needs and desires.[6]

In a classic and durable 1985 paper, Gould and Lewis enunciated three design principles, giving first place to *understanding* users and their tasks by *"direct contact from the outset."* They found many designers who thought they were doing this when in fact they were hearing or reading about them, examining user profiles, "presenting," "reviewing," or "verifying" designs with users late in the process.[7]

Implementation experience in the machine shop, at the job site, actually building the software, is just as crucial for the designer's education.

The students' needs for direct user contact and actual implementation experience argue strongly for more project courses and experiences, even at the expense of book learning. Analytical techniques and formal synthesis methods are necessary tools, but advanced methods can be self-taught when needed. Gut instincts are harder to acquire. Today's design curricula must reckon with the divorce of design and make strenuous efforts to introduce the young designer to the real worlds of implementation and use.

Notes and References

1. Ettlinger [2007], *Twinkie, Deconstructed.*
2. And listened to the sounds. I share Grady Booch's nostalgia: "I miss the sounds that old computers made. I could tell what my program was doing by the sound the computer made."
3. Codd [1959], "Multiprogramming STRETCH."
4. Kruchten [1999], "The software architect and the software architecture team." He further reports, "Some balked, sensing a waste of time, but were later amazed at how much it helped them do their jobs."
5. Mills [1971], "Top-down programming in large systems."
6. In some 22 offerings of a software engineering laboratory course, I have found it necessary and possible to solicit outside users, with whom student teams must work, and whom they must satisfy. The users have to commit time for weekly meetings with the team, in

return for which they *may* get usable prototype software. I ask for projects that would be useful if successful but not necessary. The student team must be allowed to fail.

7. Gould [1985], "Designing for usability": "These principles are: early and continual focus on users; empirical measurement of usage; and iterative design."