IEEE Software
www.computer.org/software

Up-front Design

Rebecca J. Wirfs-Brock

Vol. 25, No. 4
July/August 2008

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Up-front Design

Rebecca J. Wirfs-Brock

It is better to sleep on things beforehand than lie awake about them afterwards. —Baltasar Gracian

There can be significant benefits in thinking through a design until you get it "right enough" before launching into a major development effort. When is up-front design essential, and how can we do it more effectively? No one likes wasting time on pencil-and-paper design exercises that bear no fruit.

Build one to learn from

One of those times when careful thinking and preparation pay off is when you're designing software that breaks new ground. While our software engineering team was busy completing our first Smalltalk product, a hardware engineer in the research labs developed a color-hardware prototype. He also hacked the Smalltalk programming environment to demonstrate how his color hardware worked.

After several of us took a closer look, it was clear that his software led to more questions than answers. So, three of us spent a couple of months hashing out our design ideas for adding color to Smalltalk. No one had done color BitBlt graphics before, no one knew what it would take to enhance the existing Smalltalk system to support color, and no one had developed classes that modeled different color representation schemes. We also wanted the existing Smalltalk applications to run on the new platform without change. We were breaking a lot of new ground! Our management was easily convinced that we needed some up-front time to think about these issues before bringing others on board. We did our design thinking mostly locked in a room. We argued, sketched, waved our hands, took notes, and wrote modest amounts of simulation code. Although many details remained to be worked out after our intense design flurry, when we came out of that room, we felt confident enough to ask for more resources.

That up-front thinking let us concentrate on implementation concerns and practicalities instead of forcing innovations. Our time-crunched project involved a lot of hard work but no major design surprises.

Why did this up-front design effort pay off when many attempts fail? We were experienced developers who passionately cared about the problem. But I don't think that was as important as our tactics. We didn't try to flesh out every detail. Instead, we concentrated on a few key design areas and dug deep. We wouldn't let go until we came to working agreements (did I mention we argued a lot?). Two of us were experienced Smalltalk application developers. We were our own best, demanding customers. Management set limits. We were given freedom to explore options but felt pressure to produce results.

A more incremental approach would have been to use the prototype as the basis for evolving our design. But fortunately, the hardware engineer was open about his code's limits, and management trusted our judgment. They recognized our design team needed time to think—setting us loose to "tweak" a prototype into an innovative product would have been too risky.

 0740-7459/08/$25.00 © 2008 IEEE

Prototype to build confidence

On another project, we spent four months proving a new design approach for a system that coordinated work between telecommunications billing, provisioning, and order-taking applications. Our up-front thinking led us to develop an architecture based on a set of key design decisions. We described those decisions in a small document and implemented a prototype that demonstrated our design. Whereas the working prototype dazzled the project's stakeholders, what was important to us as a design team was that we'd proven our design ideas. Our prototype wasn't just a slick demo. Our up-front design effort gave us confidence that our design would let us predictably integrate new functionality and vendor systems.

In *The Pragmatic Programmer* (Addison Wesley 1999), Andy Hunt and Dave Thomas suggest that small, focused "tracer bullet"-style investigations are appropriate when you want to build, observe, and quickly fix your design. Tracer bullets illustrate a thin slice of end-to-end functionality. Implementing a thin slice lets you learn from small mistakes before they become big disasters.

Demonstrate progress

Even though your design might be simmering quite nicely, your progress might be intangible to others. Find meaningful ways to show others what you've been thinking about (rather than only talking or writing about your design). Give them a glimpse of what you've been doing. Ensure that your efforts are recognized and appreciated and that indeed you're working on the right things. Pulling off demonstrations, however, can be challenging—especially if you're designing something that doesn't naturally lend itself to visual representation.

I worked with a business architect who was a master at visualizing complex concepts and algorithms. Give him an afternoon, and he'd concoct a few slides that really told our story! Known as a skilled communicator, he'd jokingly start any presentation with Katsushika Hokusai's painting *Red Fuji*. Hokusai is known for painting 36 views of Mt. Fuji. Here was yet another view of our system. Such depictions (whether technically accurate or not) became important project mementos.

While I admire Hokusai's skill, I don't believe that our design illustrations require rare or exceptional talent. To be effective, they do require that we distill the essence of our work—whether it consists of code snippets, design drawings, animated storyboards, or concept illustrations—and illuminate both our ideas and thought process. Those who don't share our design context need to understand what it is that we think is important.

Develop a design rhythm

On another large enterprise application, the project team delivered feature releases two to three times a year. If a feature required innovation or investigation, they slotted it into a separate "design" track. Design track projects were monitored and given a limited time to be proven ready (or not) for prime time. Once proven, they were merged into the ongoing development stream. Some design projects never made it out of the design track for technical, political, or business reasons. Although not every worthy design project succeeded, a single design failure didn't drag down mainstream development.

How can we ensure that we strike the right balance between thinking, doing, and learning? One way is to develop a rhythm that lets design investigations get just enough ahead of development so that risks are mitigated. Lynn Miller ("Case Study of Customer Input for a Successful Product," *Proc. Agile Conf. 2005*, IEEE Press, pp. 225–234) reports how she wove user-experience design into agile development. Rather than working out all the user-experience design before the project launched, she staged it in small two- to four-week sprints. Each of these sprints preceded software development sprints by one iteration. During a user-experience design cycle, designers conducted customer surveys, did contextual inquiry, and tested low-fidelity prototypes. Once designed, the user-experience requirements fed into the next development sprint.

When you're building something new, risky, or expensive, it might seem obvious that up-front thinking and experimentation are essential. But these days, some developers shy away from even suggesting that a project might need up-front design and experimentation. They equate any up-front work with "big design up front" (BDUF) and totally wasted effort. Up-front design needn't be wasteful if you develop a design rhythm that balances thinking, learning, and doing. You'll build confidence and sleep better at night.

Find meaningful ways to show others what you've been thinking about (rather than only talking or writing about your design).

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.