



Constantine & Lockwood, Ltd.

Experience Report

Cutting Corners: Shortcuts in Model-Driven Web Design

Larry Constantine

Director of Research & Development
Constantine & Lockwood, Ltd.

Abstract: *Model-driven design under the pressure of Web-time development and impossible deadlines may require taking shortcuts, especially in which design models are developed and how. This reprint of a column describes the approach to usage-centered design taken in one crunch-mode project for a Web-deployed classroom information system. Continual access to domain experts and speed modeling with index cards are among the techniques that helped.*

Keywords: crunch mode, Web time, agile methods, lightweight methods, usage-centered design, Web applications, model-driven design, just-in-time requirements, card-based modeling

Learn more about usage-centered design at <http://www.forUse.com>.

Readers of this column do not have to be reminded of the benefits of working systematically. I have long been known as an advocate of systematic, model-driven design and development and have many times argued, in this Forum and elsewhere, that the greater the time pressure in software development, the greater is the need for thoughtful and thorough modeling. Such advice is, of course, far easier to give than to follow.

I recently completed with Lucy Lockwood one of those crunch-mode projects that tests the mettle of all involved. We were asked to design the user interface for a complex new web-based classroom application, defined by ambitious but profoundly vague requirements and being developed on a sanity-free delivery schedule that left no time for analysis, design, thinking, or sleep.

Seduced by the challenge and the opportunities to break new ground in supporting classroom teaching, we plunged in, determined to deliver a world-class design but fully realizing that there was not enough time to do the kind of thoughtful and comprehensive design models on which we have built our reputations. Had we known

Original of a column in The Management Forum, *Software Development*, February 2000. Reprinted in L. Constantine, ed., *Beyond Chaos: The Expert Edge in Managing Software Development* (Addison-Wesley, Boston, 2001). The author may be contacted at Constantine & Lockwood, Ltd., 58 Kathleen Circle, Rowley, MA 01969; tel: 1 (978) 948 5012; fax: 1 (978) 948 5036; email: larry@foruse.com | www.foruse.com

© 1999, L. L. Constantine

the full scope of the project and the void of the analysis before we signed on, we might not have done it, but then we would have missed out on a lot of fun and would not have learned some new lessons in corner cutting.

We have long argued that projects of differing size and developed on various time scales require different development processes. One-size-fits-all, “unified” methods are likely to fail on one end of the spectrum or the other. Large-scale projects may require elaborate models, meticulous record keeping, repeated validation and auditing, and careful tracing of information, while smaller, accelerated projects may need streamlined, low-overhead approaches. Web-based projects, in particular, may require stripped-down, speeded-up methods to keep pace with the demands of the rapidly evolving Internet world. [See Dave Thomas, “Web-Time Development,” this column, October 1998.]

Everyone who has worked on one of those exciting, sleep-depriving, mission-impossible projects has felt the need to cut corners, but how far do you go? When does paring down on process leave only a bare-boned skeleton inadequate to support the needs of the project?

As the late Robert Heinlein so aptly expressed it in the classic novel, *The Moon is Harsh Mistress*, there ain’t no such thing as a free lunch. Taking a shortcut always exacts a cost. Shortcutting a proven process means omitting or short-changing some productive activities, and the piper will be paid, if not now, then later.

The trick is to pick the tune and the price to pay the piper. Some shortcuts save time, while others can lead into swamplands, where backtracking can be far more costly than sticking to the straight and narrow of proper analysis and design.

Model Still

Despite pressure to deliver designs, we decided at the outset not to abandon completely the modeling we knew would help us deliver a better product. Instead, we would simplify both the models and the modeling.

Most modern software engineering, and certainly nearly all disciplined development, is model-driven to some degree. In our work as user interface designers, we use three simple models to help us understand the needs of users and fit the design to those needs. We model user roles, user tasks, and user interface contents. Associated with each of these models is a map: a role map captures the relationships among user roles, a use case map represents relationships among supported tasks, and a content navigation map represents how all the pieces of the user interface fit together. You may use more or fewer models in your work, but the odds are you use some kind of models.

Required Requirements

As the user roles for this application appeared to be neither many nor highly varied, we radically shortened the front-end modeling by constructing only a somewhat vague and admittedly inaccurate model of user roles. We were engaging in a sort of studied sloppiness, for which we knew there would be a cost, but we had little alternative. We gathered just enough information to get a good feel for the users and their ways of

working, then moved on to other matters. We never constructed a complete map of all the user roles and how they fit together.

In retrospect, this was an expensive compromise but worth the price. The heart of the matter is understanding the tasks of the users. A good user role model is a bridge to good task models and can speed up the identification of use cases in the task model, but under such tight time constraints, we concluded the payoff was not quite worth the price. Had we filled in all the blanks and crossed every tee in the role model, we might have had fewer false starts and moments of panic, but we would still not have a design.

Our advice would be to look at what models in your process serve primarily as bridges to other models rather than driving design directly. Consider cutting corners there and saving your resources for more critical steps.

Just in Case

Use cases are ubiquitous in software development today, and one particular form—essential use cases—plays a pivotal role in our work. Essential use cases represent the minimal core of capability that the user interface must provide to users, thus they not only capture basic functional requirements but also help structure the user interface around the core tasks.

Those of you familiar with use cases know there are two pieces to the use case puzzle. You have to be able to list all the use cases, and you need to describe the nature of the interaction each use case represents. In other words, to understand fully the nature of the supported tasks, you need a use case map identifying all the use cases and their interrelationships, and you need an interaction narrative defining each use case. Or, in UML-speak, you need a use case diagram for the application and a flow of events for each use case.

When it came to cutting corners in the use case modeling, we drew on our experience with larger, more disciplined projects. On one such effort, we joked that after the first hundred use cases, everyone on the team had become an expert at writing use case narratives. You reach the point where, once a use case is identified, you can almost instantly draft a rough outline of the narrative. Only for some of the more involved or exotic use cases will the interaction details not be immediately obvious to the experienced modeler.

This ability, being able to spot the occasional tough nut among the more numerous soft candies without having to bite into either, suggests one way of cutting corners in use case modeling. If your team has enough experience in use case modeling, then you may be able to skip writing the interaction narratives for many of the use cases. As you identify each use case, you make a quick but informed judgment about whether it represents an interesting or subtle problem in user-system interaction or just more of the same fairly obvious stuff.

You end up with a long list of use cases, plus narratives for some of the more interesting ones. In a pinch, this can pass for an understanding of the tasks to be supported by the system. In retrospect, this shortcut did not work quite as well as it sounds, because in the course of modeling the process narratives, you often discover

additional use cases. Thus, the shortcut can leave you with an incomplete list of use cases, which can mean missing functionality in the system.

Were we to start over, we probably would still hand-wave on many of the simpler narratives, but would push much harder on developing a complete map of all the use cases and interrelationships. Without this comprehensive map, critical functions may be discovered only late in the process, which can cost big-time in redesign. Duh.

Face It

A more successful tradeoff was substituting face-time for modeling. We were lucky to be collaborating with a team of educators who combined extensive classroom experience with advanced knowledge of theory and technique. Continuous and ready access to users and domain experts can allow designers to plug holes quickly, clarify issues on the fly, and catch mistaken notions early. It is never a recommended practice to plunge into design with incomplete requirements, but inadequate requirements models are less costly if you can simply walk across the hall to check out a design idea or talk over the cubicle wall to resolve the meaning of an ambiguous term.

Both end-users and domain experts are needed. Users are application ground-dwellers, intimately familiar with the ground covered in their jobs but relatively ignorant of important issues outside that scope. Domain experts are the hovering hawks of applications: they know the landscape as a whole but may see less of the practical details.

Easy access to users and domain experts allows requirements modeling to overlap parts of design and development. We call it “just-in-time requirements.” Where and when you need the answer to a question, you get it. In one case, a ten-minute, ad hoc conversation in the hall was enough for Lucy to pin down the requirements for two incomplete screen designs.

This game can only be played with the designers, users, and domain experts on the same playing field. If you have to play telephone tag or wait for email or schedule a meeting and drive across town to a client site, you are doomed. In fact, Lucy recognized at the beginning that the only hope for success was to work on-site, full-time with the client.

Navigating

Normally, we prefer to build an abstract prototype before we get into the final visual and interaction design. An abstract prototype has two parts: the content model, which represents how the contents of the user interface are collected for use by users, and the navigation map, which shows how all the collections are interconnected. In our experience, abstract prototyping leads to more robust, more innovative designs [see my article, “Abstract Prototyping,” this magazine, October 1998]. On this project, we chose to take the more common route and go directly from use cases into designing the layout and behavior of the user interface.

Skipping both the content model and the navigation map proved to be a mistake, and we later needed to go back to complete the navigation map before we could finish the design. The problem is that, without a navigation map showing all the screens, pages,

windows, and dialogs and how they are interconnected, you have no overview of how everything fits together. Without this picture of the architecture, you make too many mistakes in placing particular features. Once we completed the navigation map and validated it with our users and domain experts, the design process got back up to speed.

Prototypes

We also learned some lessons about using prototypes. Prototypes have many uses. At their best, they can serve as a proof-of-concept for a challenging approach or as the foundation for a sound architecture. At their worst, they can end up being shipped out as a hacked and patched substitute for a real product. In crunch-mode projects, prototypes can be a costly diversion of resources.

One problem is that prototypes are made to be thrown away, whether in whole or in part. There are reasonable arguments for building software to throw away [see Phillips, “Throw-Away Software,” this column, October 1999], but you do not want to do so unintentionally, certainly not when there is barely time to build one system.

Prototypes can allow you to get something working quickly, but don't be seduced into thinking that building prototypes will save you time. When you are caught in a time crunch, prototypes can become a major time sink. If you know what you are doing, time spent building a prototype is far better spent building the real thing. If you do not know what you are doing, building a prototype is one of the more expensive ways to find out.

Unfortunately, prototypes often serve purposes beyond software engineering. Many companies, especially start-ups, want a prototype to show off to investors and potential customers. There are many problems with such demonstration prototypes. For one thing, the better they look, the more they risk raising expectations—from customers and from management. Cobble together a slick, working VB prototype, and people will wonder why it will take months to finish the project. You may be pressured to “just clean up” the prototype and turn it into a shipping product, or you may have to explain why the prototype won't work with real data or in a networked environment.

In any case, all the time you spend putting together a demo or building a prototype is time you are not building the real thing. True, some portions of well-designed, well-constructed prototypes may be recyclable into shipped versions, but any prototype built in crunch mode is probably far too messy and fragile for much to be incorporated into the end product.

In the worse scenario, which is all too common, you not only lose time creating the prototype, but then you are expected to baby-sit it, keeping it up-to-date and ready to show to the next group of visitors being shown around.

Even paper prototypes can carry hidden costs. We use drawing tools to mock up visual designs for review, inspection, and documentation purposes. Of course, they also make great illustrations for reports, and can be turned into slides for presentations to management and ... The list goes on. You can find yourself providing and maintaining PR materials instead of solving design problems.

Teaming

Our crunch-mode project also reinforced for us the value of good teamwork. On the front-end, we had Chris Gentile and his brilliant team of educators. On the back-end, we had Larry O'Brien and his crack engineering team responsible for the programming. We were fortunate to be working with people who could quickly spot the flaws and holes in our designs or just as quickly implement a major change. When you are five hours from deadline and up to your eyeballs in interface alligators, nothing can substitute for a few good developers and one good development manager.

Learn more about usage-centered design at <http://www.forUse.com>.