Martin Fowler
fowler@acm.org

Keeping Software Soft

Do you remember when you first programmed? I remember the wonderful way in which I could type a few statements in some obtuse language into the machine, debug a bit, and the machine would respond. Not just was it easy to get the computer to do something, it was also easy to change the program to get it to do something else. It was just a matter of a few changes and a little debugging. After all, the whole point of software was to be *soft*.

Of course, the changeability of software was an illusion. In many ways, one of the biggest differences between a software developer and a regular human is that the software developers know how hard it is to change software. Or rather, it's not that it's hard to change software, it's just hard to change software without it breaking.

This problem is something that software people have known about for a long time. It is why doing ad hoc design—designing as you go about building a program—doesn't work for larger-scale systems.

To resolve this problem, software developers took inspiration from other branches of engineering. In these disciplines, there is a clear separation between the design process and the construction process. The designers communicate with the construction people by using drawings that lay out exactly how something is to be built.

This idea underpins most of the methods in software development. Instead of ad hoc design, we have what I call *up-front design*. The idea is that designers make design decisions before those decisions are programmed. The idea is that once the design is developed, it should not significantly change after programming has started. Of course, changes do occur, but the explicit purpose of the up-front process is to minimize these changes.

The difficulty, of course, is getting these designs close enough that you don't have any significant changes that cause the design to deteriorate. This is where the practical problems start appearing—it's hard to get a design right in the beginning.

The Reality of Changing Requirements

But there is a more fundamental problem. From time to time, I have to visit badly messed-up projects, to see if there is a way out of trouble. Whenever this occurs and I talk to the developers, I always seem to get the same lament: "The users keep changing the requirements." This always surprises me: I'm surprised *anyone* is surprised by changing requirements. I've never come across a serious project where requirements don't change. You can get users to sign off on requirements documents in every blood group, but they still change their mind.

They change their mind for several reasons. One is that it is very hard to visualize what a computer system will look like. Only when you have it in front of you and start using it for real work do you find out what really is useful and what isn't. Another is that businesses change, and a very real and important requirement six months ago can become minor as a business changes. Exacerbating this is the fact that users increasingly have done a little programming themselves: some VB here, an Excel macro there. So they know that it's easy to change software (they never get systems with the dependency problems that professionals deal with).

For many, this is the challenge of requirements engineering. We need to be better at getting requirements right in advance, to spend more time to get them right, to come up with new techniques. I used to believe this too, but now I've come to the conclusion that this is unreachable, at least for a while.

One of the key issues here is the balance of *wants* and *costs*. If you buy a car and the salesman says, "Would you like a sunroof," the first question you'll ask is, "How much?" For $20 I would have it fitted, for $20,000 I'd do without. We cannot expect our users to fix their requirements until we can estimate our costs with reasonable accuracy. Unfortunately, we are very bad at cost estimation. I don't think this is because software developers are stupid; the problem is that our basic materials keep changing so quickly. Civil engineers would find it difficult to estimate if the fundamental properties of concrete kept undergoing a major "upgrade" every year.

So what's the alternative? I think it comes down to expecting requirements to change and using a development process that relishes change. This strikes at the core of the up-front design process. If requirements change without warning, how can we develop a stable up-front design?

Making Change Easier

The answer is to build software to make it more able to deal with unanticipated change. (We can build a design that caters for anticipated changes—it is the unanticipated ones that bite you in the bum.) Objects are a key part of this. By using objects with encapsulation and poly-

Martin Fowler is an independent consultant based in Boston, Massachusetts.

morphic interfaces, we can improve the packaging of our software, thus reducing the dependencies and making things easier to change.

Dynamic development environments also help with browsers, quick ways to get at cross-references, debuggers and inspectors that let you explore code, and rapid turns of the compile/link cycle. For many professional developers, these tools are still new, but any Smalltalker knows how much difference it makes when you can edit code in a debugger and have the change made instantly, without an attention-breaking gap while compiling and linking go on.

Refactoring plays a big role here too. Refactoring is a set of techniques that let you change the design of software efficiently without introducing bugs. With refactoring, if you get a design wrong, you can change it later without incurring a huge cost.

Dynamic Design All of these techniques point toward a shift in the way we can go about doing design, moving toward what I call *dynamic design*. With dynamic design, you don't try to get the design right at the beginning. This doesn't mean you abandon up-front design; you still do it, but now you don't try to find the *best* solution. Instead all you want is some *reasonable* solution. You know that as you build the solution, as you understand more about the problem, you will realize that the best solution is different from what you originally came up with. With refactoring, objects, and a dynamic environment, this is not a problem, for it is no longer expensive to make the changes.

An important result of this change in emphasis is a greater movement toward simplicity of design. Before I used dynamic design, I was always looking for flexible solutions. With any requirement I would be wondering about how that requirement would change during the life of the system. Because design changes were expensive, I would try to build a design that would stand up to the changes that I could foresee.

The problem with building a flexible solution is that flexibility costs. Flexible solutions are more complex than simple ones. The resulting software is then more difficult to maintain in general, although it is easier to flex in the direction I had in mind. However, you have to understand *how* to flex the design. For one or two aspects this is no big deal, but changes occur throughout the system. Building this flexibility in all these places makes the overall system a lot more complex and expensive to maintain.

The big frustration, of course, is that all this flexibility is not needed. Some pieces of it will be, but it's impossible to predict which. So to gain the flexibility you need, you have to put in a lot more flexibility than you need.

With dynamic design, you approach the risks of change differently. You still think about potential changes, you still consider flexible solutions. But instead of implementing these flexible solutions, you ask yourself, "How difficult is it going to be to change a simple solution into a flexible solution?" If, as happens most of the time, the answer is, "Pretty easy," then you just implement the simple solution.

So dynamic design can lead to simpler designs, without sacrificing flexibility. This makes the design process easier and less stressful. Once you get a broad sense of those things that refactor easily, you don't even think of the flexible solutions. You have the confidence to refactor if the time comes. As Kent Beck advises, you build the simplest thing that could possibly work. As for the flexible, complex design, most of the time you aren't going to need it.

That's quite a change in design process, and it's been a big shift for me as a designer. It has several preconditions. You need good tests, you need objects, you need to know how to refactor. But the reward is the ability to put away the fear of changing requirements and to be responsive to your users without sacrificing your design, and thus your future.

CHEVY NOVA AWARDS

Online OFFLINE

Here are the nominees for the *Chevy Nova Award,* given in honor of GM's fiasco in trying to market the car in Central and South America (in Spanish,*"no va"* means "it doesn't go."

- The Dairy Association's huge success with the campaign "Got Milk?" prompted them to expand advertising to Mexico. However, it was soon brought to their attention the Spanish translation read "Are you lactating?"
- Coors put its slogan, "Turn It Loose," into Spanish, where it was read as "Suffer from Diarrhea."
- Scandinavian vacuum manufacturer Electrolux used the following in an American campaign: "Nothing sucks like an Electrolux."
- Clairol introduced the "Mist Stick," a curling iron, into Germany, only to find out that "mist" is slang for manure. Not too many people had use for the "Manure Stick."
- When Gerber started selling baby food in Africa, it used the same packaging as in the US, with the smiling baby on the label. Later they learned that in Africa, companies routinely put on labels pictures of what's inside, since many people can't read.
- In China, Pepsi's "Come alive with the Pepsi Generation" translated into "Pepsi brings your ancestors back from the grave."
- The Coca-Cola name in China was first read as "*Kekoukela,*" meaning "bite the wax tadpole" or "female horse stuffed with wax," depending on the dialect. Coke researched 40,000 characters to find a phonetic equivalent "*kokou kole,*" translating into "happiness in the mouth."
- Frank Perdue's chicken slogan, "It takes a strong man to make a tender chicken" was translated into Spanish as "it takes an aroused man to make a chicken affectionate."
- When Parker Pen marketed a ball-point pen in Mexico, its ads were supposed to have read, "It won't leak in your pocket and embarrass you." The company thought that the word "*embarazar*" (to impregnate) meant to embarrass, so the ad read: "It won't leak in your pocket and make you pregnant!"
- When American Airlines wanted to advertise its new leather first class seats in the Mexican market, it translated its "Fly in Leather"campaign literally, which meant "Fly Naked" (*vuela en cuero*) in Spanish.