Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

The State of Design

Martin Fowler

This is the last Design column that will appear in *IEEE Software* under my editorship. I'm passing the reins over to Rebecca Wirfs-Brock (www.wirfs-brock.com), whose writing has helped me ever since I started in this business. So, as I say goodbye, this seems like a logical time to reflect on what's appeared in this column over the past five years. Let's see if I can tie all the articles together into a coherent story.

Design principles

A useful way to talk about design is to identify principles that lead to well-designed systems.

My first column ("Avoiding Repetition") discussed probably my favorite design principle. (Throughout, I include each column's title in parentheses; see the sidebar for authors and issue.) This principle is relatively easy to understand, yet applying it in a determined effort to remove duplication in code often leads to effective designs. Ideally, all principles should be like this—simple to express yet complex in their consequences.

Unfortunately, it's hard to find and capture good design principles. One topic I always intended to cover was orthogonality—ensuring that different things are kept separate, or that each part of a system has a single and clear set of responsibilities. Although this is a good principle, I find it hard to apply. How do you decide how big the bounds of a cohesive module should be? What exactly is a single set of responsibilities? Fortunately, sometimes a general principle such as this can spawn more concrete principles that are easier to apply (and easier to write about). A good example of this is separating user interface code from other system parts ("Separating User Interface Code").

Modules and interfaces

A central design idea is to break a system into modules that hide complex implementations behind clear interfaces. A good way to make these divisions is to view each piece as a protected variation ("The Importance of Being Closed"), identifying system areas that change and then hiding that variation behind an unchanging interface. To give these modules good interfaces, you need to make the interface easy for other programmers to use ("The Most Important Design Guideline?"). This idea of variations behind interfaces is the essence of encapsulation, which goes far beyond just hiding data—although it's important to have good data access routines when you need them ("Data Access Routines").

Interfaces are handled differently depending on what they're separating. Object-oriented systems are well known for decomposing systems into small objects with interfaces, but we must treat these kinds of interfaces differently from those that are exposed across a network or shared between teams. Not all interfaces are clearly recognized by languages or design notations—I've long argued that the difference between published and public interfaces ("Public versus Published Interfaces") is extremely important. As you produce these larger-grained systems, you also need to worry about how the components will interact ("Components and the World of Chaos") as well as how to assemble them without coupling them during construction ("Module Assembly"). Large components often figure most in design discussions, but don't forget that you can gain a lot by defining smaller objects that simplify the larger structures ("When to Make a Type").

Interfaces help separate modules in a program's text, but increasingly we find value in separating modules by time as well. Most developers are used to synchronous behavior, where a caller waits for a return before continuing. However, many systems, like baristas, gain by thinking asynchronously ("Your Coffee Shop Doesn't Use Two-Phase Commit").

Fitting into the development process

A significant issue in software design is determining where design fits in the overall software process. Is design a development phase that should be (mostly) done before programming, or should we intermingle it with programming? I've made little secret of my preference for the latter—with the implication that design is a continuous process ("Continuous Design"). Thinking about design in this way has its own challenges and rewards, which haven't been sufficiently explored. Fortunately, the agile meme's proliferation has opened up this discussion.

If you agree that design and programming are intertwined, then the program you write has a special value—it also acts as vital design documentation. As a result, many programming issues related to writing code are also design decisions. I've always emphasized writing clear code that's easy to understand, which has led me to value explicit code ("To Be Explicit"). I often hesitate to use a mechanism that doesn't make clear what it's doing. Explicitness is also valuable at runtime, which is why I favor loud and rapid failures in response to errors ("Fail Fast"). In either case, design approaches are closely tied to our programming languages—language constructs often affect design decisions ("How .NET's Custom Attributes Affect Design").

However, explicitness in code has its limits. System behavior sometimes comes with enough variation that explicit code gets too repetitive. When principles of explicitness and of avoiding repetition start clashing, I prefer to avoid repetition. That's when metadata techniques come into their own ("Using Metadata"), using reflection or code generation. We can use this kind of metadata approach for such things as decision tables and simple spreadsheets ("Design to Accommodate Change"), which can combine ease of change with a greater visibility to domain experts who aren't professional programmers.

Another design consequence related to agile methods is an emphasis on testing. Agile methods view testing as part of design, not an afterthought. Test-driven development ("Aim, Fire") has been very popular in the circles I inhabit. It's a powerful design aid because it forces you to think about interfaces before implementation. Of course, it also helps people develop good regression tests, which

Continued on p. 16

Five Years of Design

Following is a list of all the Design columns, which you can download from http://martinfowler.com/articles.html#id109573 or www.computer.org/software.

2001

"Avoiding Repetition" by Martin Fowler, Jan./Feb.
"Separating User Interface Code" by Martin Fowler, Mar./Apr.
"The Importance of Being Closed" by Craig Larman, May/June
"Reducing Coupling" by Martin Fowler, July/Aug.
"Aim, Fire" by Kent Beck, Sept./Oct.
"To Be Explicit" by Martin Fowler, Nov./Dec.

2002

"Modeling with a Sense of Purpose" by John Daniels, Jan./Feb.
"Public versus Published Interfaces" by Martin Fowler, Mar./Apr.
"Yet Another Optimization Article" by Martin Fowler, May/June
"How .NET's Custom Attributes Affect Design" by James Newkirk and Alexei Vorontsov, Sept./Oct.
"Using Metadata" by Martin Fowler, Nov./Dec.

2003

"When to Make a Type" by Martin Fowler, Jan./Feb.
"Patterns" by Martin Fowler, Mar./Apr.
"Components and the World of Chaos" by Rebecca Parsons, May/June
"The Difference between Marketecture and Tarchitecture" by Luke Hohmann, July/Aug.
"Who Needs an Architect?" by Martin Fowler, Sept./Oct.
"Data Access Routines" by Martin Fowler, Nov./Dec.

2004

"Continuous Design" by Jim Shore, Jan./Feb.
"Module Assembly" by Martin Fowler, Mar./Apr.
"MDA: Revenge of the Modelers or UML Utopia?" by Dave Thomas, May/June
"The Most Important Design Guideline?" by Scott Meyers, July/Aug.
"Fail Fast" by Jim Shore, Sept/Oct.
"Before Clarity" by Michael Feathers, Nov./Dec.

2005

"Your Coffee Shop Doesn't Use Two-Phase Commit" by Gregor Hohpe, Mar./Apr.
"Design to Accommodate Change" by Dave Thomas, May/June
"The Test Bus Imperative: Architectures that Support Automated Acceptance Testing" by Robert C. Martin, July/Aug.
"Enterprise Architects Join the Team" by Rebecca Parsons, Sept./Oct.
"State of Design" by Martin Fowler, Nov./Dec.

Just because artifacts are linked doesn't mean that a change will propagate. Engineering judgment is necessary for determining which impact-tree branches can be pruned or where you must add new branches because new artifacts are required. Traceability rationale can help you determine the precise nature of change propagation.

4. *Define change.* Traverse the impact tree, working out the precise details of the changes at each point. A configuration management tool can help you do this.
5. *Apply change.* When the changes are ready, apply them to the system in all affected layers.

At each stage, you'll gather more precise information about the nature of the change, including cost. A go/no-go decision point can follow each stage.

Whatever development scale I engage in, I systematically apply information traceability. It's a vehicle for thinking about the way the software meets its requirements; it captures design rationale to help others understand and review; and it gives me far greater confidence in managing future changes.

Reference

1. E. Hull, K. Jackson, and J. Dick, *Requirements Engineering*, 2nd ed., Springer, 2004.

Jeremy Dick is a principal analyst at Telelogic UK. Contact him at jeremy.dick@telelogic.com.

Continued from p. 13

are essential when evolving the design. Indeed, many view testability as a vital design property, particularly with older systems ("Before Clarity"), leading to design architectures that make the systems more testable ("The Test Bus Imperative").

Designers are people too

Design involves people, so in addition to considering how design fits into a process, you also have to think about how it fits with an organization's people. A common question is, What's the difference between architecture and design?—which raises the question of what an architect's role is ("Who Needs an Architect?"). People often view architects as holding a separate, directing role. However, I strongly believe that technical leaders should work closely with the developers on a team, a principle that also applies to enterprise-wide architects ("Enterprise Architects Join the Team").

This issue of design leadership goes further. Architecture can be about the technical software structure as well as about how the software faces its users—leading to questions that many technical architects don't consider as frequently as they should ("The Difference between Marketecture and Tarchitecture").

Representing design

For a large part of my career, people have talked about representing design in terms of notations, particularly graphical notations that try to tell you important things about a program's structure. As someone who has written books on one of these, I understand both the capabilities and limitations of graphical notations. A common problem with using these notations is that people use them to represent different kinds of perspectives—even for a single system. Think of three primary purposes for these models: conceptual, specification, and implementation ("Modeling with a Sense of Purpose"). I've come to the conclusion that models are useful for certain tasks, such as class structure or visualizing dependencies ("Reducing Coupling"). However, I don't see them as absorbing the future of software development ("MDA: Revenge of the Modelers or UML Utopia?").

Few of these design principles and discussions are new. It's long been known that you should avoid premature optimization ("Yet Another Optimization Article"), yet constantly we see people doing just that. This is why I spend so much energy simply trying to find good design techniques that have worked well in the past and trying to explain them to others so they'll use them in the future ("Patterns").

So there it is—five years of writing compressed into a single column. I hope that my various authors and I have given you a few useful ideas along the way, and I'm pretty certain that Rebecca will find a lot of good material and be an excellent steward of this column. I will, of course, continue to write on my Web site (http://martinfowler.com), and I hope to continue finding useful ideas to write about.