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

Continuous Design

Jim Shore

The rising popularity of refactoring, tools such as JUnit, and agile methodologies such as Extreme Programming (XP) has brought a new style of design into view. *Continuous design* is the process of using refactoring to continuously improve a program's design. Initially a skeptic, I've been experimenting with continuous design for four years, and it's changed the way I program.

The role of continuous design

Continuous design is also known as evolutionary or emergent design. I prefer the term *continuous design* because it emphasizes the core of the process: continuously taking advantage of opportunities to improve your design. When you discover a design flaw, you fix it. When a new feature doesn't fit, you update the design. (For an introduction, see Martin Fowler's "Is Design Dead?" at www.martinfowler.com.)

Continuous design can coexist with up-front design, but the XP approach advocates the former instead of the latter. When I started experimenting with XP in 2000, I was very skeptical of the idea. I'd been involved in projects that had painted themselves into a corner with bad design, and I was convinced that up-front design was necessary to avoid that problem. When I tried XP, I hedged my bets: I designed a layered architecture and persistence model up-front.

The persistence model was a disaster: it required a huge amount of code to do simple things. Continuous design, however, was a roaring success. The application, developed over 16 months by six people, had the best design I'd seen. Bit by bit, continuous design fixed the persistence model too, eventually giving us an elegant solution that was simple and flexible.

My initial success with continuous design inspired me to continue experimenting. My up-front designs became simpler and simpler ... then disappeared entirely. Today, when I start a project, I actively avoid deciding on a design. (That's harder than it sounds!) I implement the first feature, see where it takes me, implement the second one, and refactor. As someone who used to strongly advocate up-front design, I'm surprised by how successful this has been. My designs are actually simpler and easier to extend than they were with up-front design.

It's easy to take a statement like "I actively avoid deciding on a design" to mean that I don't design at all. That's not what I mean. Continuous design involves intensive, constant review of the design—after code has been written and there's a design to review. The difference is that there's no speculative design. You optimize the design for the features you've already coded.

Hard problems

How far can you take continuous design before it breaks down? Some decisions are difficult to change. Even enthusiasts of continuous design will cite "hard problems" such as security, transactions, and internationalization that require up-front design.

Or do they? The line isn't as clear as people think. I've solved each of these problems using continuous design. The better the application met continuous design goals (see the accompanying sidebar), the easier the change was.

Security

The first time I encountered a pervasive retrofit was in adding transaction security to an existing system. By transaction security, I mean that every client action in this client-server system was separately verified to see if the client had permission to do it. This level of security was a critical requirement, but a for-

 0740-7459/04/$20.00 © 2004 IEEE

mal requirements process had failed to unearth it. (This was despite having key stakeholders review a prototype, attend feature prioritization meetings, and sign off on a 76-page requirements document, complete with screen shots.) After the first version was released, my team was brought back to add the missing security.

This wasn't an XP project: we had undergone an up-front design phase. We hadn't anticipated transaction security, though, and the design didn't include anything to handle it. Security still turned out to be easy to add, although it was tedious. We added security status objects that we passed around and added security checking to every server-side entry point. There were quite a few, and updating the code took three programmers about four days: nontrivial, but not as significant as a pervasive change such as this would lead you to expect.

Transactions

A year or so later, I retrofitted business transactions into a Web-based application I was leading. Business transactions, in this case, meant the ability for operations on the business layer to be grouped into atomic transactions, to be committed or rolled back as a unit.

This project had been developed with XP and had very good code. This retrofit, which was more significant than the security retrofit, took much less time because the code was better. It took one pair about a day, maybe two. We took an existing database connection manager object, renamed it `Transaction`, and added atomicity features. Updating centralized database and exception-handling logic rounded out the change.

A notable aspect of this change was the centralized database handler. Most applications have database connection management sprinkled throughout the code. Even when there's a dedicated persistence layer, there's often repeated connection-open, connection-close, and exception logic. In the beginning, we had duplicated connection logic, too. That duplication was a target for continuous design, and we refactored it into a common method on the `Connection` object. Business objects would pass the `Connection` object to the persistence layer, which would give it a block of code to execute.

This design, made to eliminate duplication, is what made creating `Transaction` so easy. We changed `Connection` to store the code blocks and then created a `commit()` method that executed them.

As you can see, our design was simple but not simplistic. Using code blocks is a sophisticated technique in Java, the language we used. We stay up to date with the latest design patterns and techniques—and we do our best to avoid needing them. This time, we did need them.

Internationalization

About six months later, we retrofitted the same application with internationalized input and output. The code had been developed with continuous design for nearly a year and was simply outstanding. After researching internationalization, it took one pair about four hours to code it. The code had a centralized user input processing method, which required a five-line change. The centralized output handler required a 10-line change.

It might seem like we'd planned up-front for these changes, but we didn't. Early in development, we had the same two lines of user input handling duplicated in three or four places. Improvements to our unit tests required it to be centralized, so we refactored. Months later, when we had dozens of pages, we came back and reused that method to support international input.

Output was a similar story. We originally used an HTML templating library called WebMacro (www.webmacro.org) as a framework that our classes extended. This framework-based approach forced design decisions that caused a lot of duplication. We didn't like this, so we moved to an approach that isolated WebMacro in a wrapper class. This required us to work harder to understand WebMacro, but it made our design much cleaner.

The end result of this change was a single class that passed an HTML template to WebMacro and returned the resulting stream. Later, when we internationalized, we were able to easily modify that class to choose between several localized templates.

Internationalization, again

If we'd kept WebMacro's framework-based approach, internationalization would have been a lot more difficult. In fact, it probably would have been like our recent internationalization of a C# ASP.NET application. ASP.NET is a framework for .NET Web applications. Its use of multiple base classes and data-driven components makes it difficult to factor out

Design Goals in Continuous Design

Continuous design makes change easy by focusing on these design goals:

- **DRY (Don't Repeat Yourself):** There's little duplication.
- **Explicit:** Code clearly states its purpose, usually without needing comments.
- **Simple:** Specific approaches are preferred over generic ones. Design patterns support features, not extensibility.
- **Cohesive:** Related code and concepts are grouped together.
- **Decoupled:** Unrelated code and concepts can be changed independently.
- **Isolated:** Third-party code is isolated behind a single interface.
- **Present-day:** The design doesn't try to predict future features.
- **No hooks:** Interfaces, factory methods, events, and other extensibility "hooks" are left out unless they meet a current need.

common functionality. Combined with our inexperience with ASP.NET, this led to a lot of duplication in the presentation layer.

We only internationalized output, not input, but it still took one pair about a week and a half. The culprit was duplication. Rather than change one method that handled all input and output, we had to find all places that output was generated and change that. As with the security retrofit, this was tedious, but not particularly hard.

Why does it work?

On all these projects, the difficulty of making changes directly related to specific design qualities. The most obvious is duplication: when a change we wanted was localized in a single class, it was trivial. When we had to modify similar code over and over, the change was tedious and took a lot longer.

Other design qualities also affected our ability to make changes. Simplicity was important. With simpler designs, we were less likely to encounter existing code. When adding features, we were better off when there was no preexisting design to handle that feature. Adding code that doesn't exist is easy; fixing someone's preconceptions about a feature first is more costly. The sidebar lists a number of other design qualities that have made our projects easier to maintain and change.

Before you try

My experiments with continuous design have been very successful. I recommend that you try it on your projects. Before you begin, though, look at your current process. Software processes oriented around up-front design might not be friendly to continuous design. At a minimum, you'll need automated tests, a team-based approach to changes (such as collective code ownership), and commitment to continuously evaluating and improving your design in the face of schedule pressure.

You might wish to experiment with continuous design by mixing it with up-front design. If you do, be aware that continuous design requires specific design goals (see the sidebar). In particular, up-front designs often include "extensibility hooks" for future design changes. This approach makes continuous design harder and should be avoided.

On my projects, continuous design's focus on simplicity and continuous improvement has made the code better and more maintainable over time, rather than less. After experimenting with continuous design for so long, I'm convinced that it's harder to paint yourself into a corner than it is with up-front design. Try it for yourself, and let me know how it worked for you.

Jim Shore is the founder of Titanium I.T., a Portland, Oregon, consultancy specializing in Extreme Programming. He'd like to hear about your experiences with continuous design. Contact him at jshore@titanium-it.com.

Continued from p. 19

installation. But any task that a developer has to perform more than three times is a good candidate for automation.

Implementing automation can be as simple as writing a shell script or batch file, or a macro in your integrated development environment. Or, you might add additional rules or targets to an existing build script (such as Ant or Make would use). It might require an entire program itself, written in Ruby or Java. However it's implemented, make sure that the automation code is kept in version control and is advertised and available for the entire team's use.

Automation gives the team consistency, reliability, and repeatability across different developers and environments. New developers can get on board and be productive much faster if all they have to do is push a button or type a command, even if they're building on a different platform than they're accustomed to.

Products to coordinate compilation and building include old standbys such as Ant (ant.apache.org) or Make (www.gnu.org/software/make). Systems such as AntHill (www.cs.unibo.it/projects/anthill/index.html), Cruise Control (http://cruisecontrol.sourceforge.net), or Dartboard (http://public.kitware.com/Dart) perform continuous build and integration.

With these three legs in place, you'll have a firm base from which to build great code.

Andy Hunt and Dave Thomas are partners in The Pragmatic Programmers and authors of the new *The Pragmatic Starter Kit* book series. Contact them via www.PragmaticProgrammer.com.