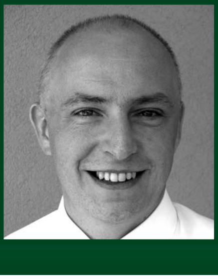
point



Design Patterns Are Bad for Software Design

Peter Sommerlad, *IFS Institute for Software, HSR Rapperswil*

Design patterns make it too easy to introduce unnecessary complexity into system design and are much too often applied without discipline or experience.

I feel guilty as an author of many patterns and a pattern community supporter because I've come to the conclusion that—in general—design patterns are bad for software design. It's true that the Gang of Four launched a cultural revolution through their splendidly successful book (*Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994), catapulting OO design and programming into the mainstream. However, in the early days of OO programming, only software gurus designed working OO systems—the average programmer was stuck with Basic, Pascal, or C.

The gurus invented the architectures that later became popular patterns, and they either consciously thought through their design decisions or made gut decisions based on significant experience. In addition, they came from a time when the principles of simplicity, abstraction, structure, encapsulation, coherence, and decoupling promulgated by the likes of Edsger Dijkstra, Tony Hoare, and David Parnas were well known, taught, and followed—at least by the people considered capable of good software design.

Today, design patterns let average developers design working OO systems that would have otherwise been beyond their capabilities. This might sound like a great thing, but the relative lack of expertise or brilliance can easily result in bigger software design disasters than would occur without them. For example, most GoF patterns are about introducing flexibility by indirection and inheritance. This is great when you use it to reduce code size and simplify logic by applying polymorphism, but it can also be a tool for overengineering and introducing excessive complexity. A designer who can't decide on a system property can use design patterns to postpone decisions, speculate about features never needed, and lay a heavy burden on system implementers and maintainers.

When teachers who are inexperienced in OO programming use only the GoF book to train and educate their students in OO design, the problem becomes institutionalized in the profession. The GoF book seldom explains a pattern's drawbacks well. Readers or teachers who fail to consider these drawbacks can easily see patterns as an OO design panacea.

The GoF presentation format contributes to some of the confusion. Most modern pattern books have adopted a format that more clearly shows the problem, the forces resolved, and the solution's potential downsides. If the GoF book is your only source for patterns and OO designs, you're working with outdated material—not only because of its aged pattern format

Continued on page 70

0740-7459/07/$25.00 © 2007 IEEE

Every Good Designer Uses Patterns

James Noble, *Victoria University of Wellington*

About halfway down the first page of *Design Patterns*, the authors ask a question: "Experienced designers evidently know something inexperienced ones don't. What is it?" This isn't a new question, nor is it unique to software. What do chess grandmasters know about chess that schoolchildren don't? What do old masters know that their apprentices don't? What do the engineers of the iPod, 747, or the InterCityExpress know that their unsuccessful competitors don't?

The answer is more than the "rules of the game," the fundamental mathematical or physical principles that underlie the engineering disciplines, market research, consumer psychology, or luck. For a great design, all these are necessary but not sufficient.

The extra ingredient—and, of course, the answer to the question posed in *Design Patterns*—is an understanding of past designs, past practice, which designs have succeeded, which designs have failed, and why. So, a chess grandmaster will understand the attacks, defenses, and patterns of play; aspiring artists sit in front of old masters in galleries or studios and copy their brushwork; and engineers study key innovations from existing designs and incorporate them into their own work.

And, quite simply, this is what the 23 patterns in the Gang of Four's landmark book managed to do. They captured emerging "best practice" for solving particular design problems. For example, Iterator, Observer, and Composite are all built into the designs of industry-standard libraries, and you can't be a competent programmer without understanding them. These patterns aren't just about extensibility; they're about object-oriented modeling. If you have to loop over the elements of a stream or a collection, you'll use an Iterator. If you have to implement a tree of self-similar objects, you make a Composite. And you'd better have very good reasons to do things another way. The patterns have become so common throughout the industry that they constitute a shared vocabulary among OO programmers.

As a result, the 23 original patterns—and the wide range of patterns they inspired—have become part of computer science's core knowledge alongside Tony Hoare's "Notes on Data Structuring" in *Structured Programming* (Academic Press, 1972) or Donald Knuth's catalogs of algorithms in *The Art of Computer Programming* (Addison-Wesley, 1997). Just as you can't call yourself a computer scientist (or an expert programmer) if you don't know what a Linked List or a Binary Search is, these days you need to know an Iterator, too.

Partly this is because patterns are intrinsically small, modest, and tied to particular problems, contexts, and examples.

Patterns are crucial to the art and science of software design and programming, rooted in hard-won practice and experience.

Continued on page 70

0740-7459/07/$25.00 © 2007 IEEE

point continued from page 68

but also because the subsequent pattern literature offers better solutions to design problems.

The paramount example of an obsolete pattern that introduces more problems than it solves is the classic Singleton. Rather than use the limited space here to argue against it, I prefer to quote Kent Beck: "How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead" (*Test-Driven Development: By Example*, Addison-Wesley, 2002, p. 179).

The key word in this quotation is "think." Patterns should make you think, but the GoF design patterns often allow developers to avoid thinking about design because implementing patterns is so easy, especially Singleton. And getting rid of them is so hard. This asymmetry between thought and implementation ease leads to design-pattern accumulations in systems, even when they're not needed and get in the way of improvement or simplicity. I've seen people refactoring a simple "if" statement into State, thus increasing code size and complexity for no reason.

Whenever designers feel the itch to apply a design pattern, I would ask them first to think. Do they really need its flexibility, and will adding the pattern's complexity make the overall system simpler? In addition, I'd ask them to unlearn some patterns, such as Singleton, and relearn, understand, and apply the basic principles of good design such as simplicity, low coupling, and high cohesion.

Nevertheless, I should be happy. Design patterns led to many OO legacy systems that need refactoring or reengineering, and I get called to help with that task. One system, for example, contained hundreds of Singleton classes, completely destroying modularity. Please, dear developers, learn more than 23 GoF design patterns and think before you apply them. They are much harder to get rid of than to add later if needed.

Peter Sommerlad is professor for software engineering and head of the IFS Institute for Software at HSR Hochschule für Technik, Rapperswil, Switzerland. He's coauthor of *Pattern-Oriented Software Architecture*, vol. 1, and *Security Patterns*. His current research topic is "decremental development," focusing on refactoring for non-Java languages such as C++, Ruby, Javascript, Ada, and Python in Eclipse. Contact him at peter.sommerlad@hsr.ch.

continued from page 69 counterpoint

Many other recent approaches to software design—methodologies such as the Rational Unified Process, notations such as the Unified Modeling Language, and technologies such as computer-aided software engineering and Model-Driven Architecture—require programmers to make global commitments to an overarching design theory that then dictates uniform solutions to whatever problems arise. In contrast, patterns are contingent and partial. They don't come with the baggage of a big story about what programming is. Rather, a pattern gives a concrete solution to a single, local problem. It considers the tradeoffs and negotiation between the forces that arise in the program's actual context—for almost any methodology or technology programmers must face.

Empirical evidence suggests that expert programmers think about their programs in ways that correspond to design patterns. For example, Elliott Soloway and Kate Ehrlich's work on programming with clichés back in the early 1980s ("Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng*. vol. 10, no. 5, 1984, pp. 595–609) showed that even novice programmers know rudimentary patterns to do with looping and variable assignments. John Carroll and his colleagues showed that the best way to present knowledge about design or interaction tasks is in small, individual pieces that give concrete solutions to individual problems (*The Nurnberg Funnel*, MIT Press, 1990).

Because hundreds of thousands of programmers have used the GoF's *Design Patterns* since 1994, we know a lot more about patterns now. The Pattern Language of Programs conferences held around the world each year since 1994 have introduced many new good patterns, and patterns researchers have compiled many collections of both general and special-purpose patterns. The earliest patterns books—*Design Patterns* and *Pattern-Oriented Software Architecture*—continue to sell well and inspire more recent patterns on topics from Java to J2EE, .NET to interaction design, and dating to software architecture.

The essence of software design is finding solutions to problems in contexts with many conflicting forces. It involves considering each solution's consequences, benefits, and liabilities, then making engineering decisions based on concrete, proven experience. Patterns are the best introduction to practical software design for programmers serious about their craft.

James Noble is a professor of computer science and software engineering at Victoria University of Wellington, New Zealand. His research centers on software design, ranging from object orientation, aliasing, design patterns, and agile methodology to postmodernism and the semiotics of programming. Contact him at kjx@mcs.vuw.ac.nz.

Peter Responds

James is right in almost everything he wrote, and certainly I expressed my point in some exaggeration. Nevertheless, his title says it: “good designer.” Today, the majority of software creators aren’t. A multitude of reasons account for this situation—I won’t try to list them all. However, for example, many developers today have no formal training in software engineering, and popular technology courses offer only short-term value. Active code reading isn’t part of software education either; and in the rush for features, proactive maintenance (refactoring) and learning from existing code is neglected.

As with any profession, continuous learning is essential for being “good” at software design. I believe patterns are a key ingredient for continuous learning, but we educators fall short of reaching average developers.

Design patterns should help create simpler solutions, but they can easily generate complexity instead. Computer speed and space have increased so much that the need for simple, elegant solutions is no longer as obvious as it was when these resources were scarce. But the brains creating software haven’t kept pace with Moore’s law. Simpler solutions are easier to understand but harder to create. Simplicity in design is often neglected.

James mentions the pattern style of expert thinking. I would like to emphasize that design patterns form the language of software design experts. But what language proficiency do you have, when your vocabulary consists of only 23 words…er…patterns?

James Responds

It’s a strange honor to debate Peter about the usefulness of design patterns. I’ve long admired his patterns work, finding both *Pattern-Oriented Software Architecture* and *Security Patterns* insightful and practical in equal measure. But his arguments here can’t go unanswered. Yes, the Gang of Four’s book marked a cultural revolution in software design. But blaming over-engineered, overly complex, or overtly pretentious object-oriented designs on the patterns movement, the way patterns are taught, or the *Design Patterns* book itself is going several steps too far!

Peter’s key argument is that patterns help working programmers produce better designs than they could have otherwise. It’s hard to see this as any kind of failure. Does it mean that programmers will stop thinking about what they’re doing? Of course not!

Abstraction, simplicity, structure, and all the rest—although they’re fine principles for software development—aren’t absolutes. Applying principles to practical designs is difficult, and patterns help programmers think about designs. Furthermore, Joshua Kerievsky’s *Refactoring to Patterns* (Addison-Wesley, 2004) shows how to add patterns incrementally to programs when required—and how to remove them when they’re no longer necessary.

Will inexperienced teachers teach patterns badly? Of course, and experienced teachers who are trained in theory and well-versed in practice will offer students a deeper understanding of OO design and the patterns it produces. Still, OO design does produce patterns, and even inexperienced teachers will do better with pattern books than without them.

Sure, more recent books have adopted different formats for presenting patterns; and some patterns, such as Singleton, might be less applicable now then they were 12 years ago. But these are minor issues that we can address with more experience, experimentation, and education using patterns—not less.