Principles Versus Patterns

Carlo Pescio, Eptacom Consulting

Over two decades ago, expert systems were one of the most promising technologies. Then researchers rapidly stumbled into what's called the Feigenbaum Bottleneck: As domain complexity grows, it becomes very difficult for human experts to formulate their knowledge as practical strategies (as human "say-how"). It is easier to demonstrate knowledge by doing ("show-how"), and most people can also make good choices between alternatives ("say-what").

Not much attention has been paid to the Feigenbaum Bottleneck outside the domain of artificial intelligence. This is unfortunate, because the bottleneck hints at the inner nature of human beings. To see the depth of its effects, we have to look no further than the current state of software systems design.

PRINCIPLES, PRINCIPLES EVERYWHERE

Early in the history of programming, brilliant people realized that every good software system has some desirable properties: It should be extensible; parts of it should be modifiable without major impact on other parts; and so on. Because of the Feigenbaum Bottleneck, it is very hard to describe precise, step-by-step instructions to build systems with such properties. It is easier to articulate the desirable properties in the form of design principles.

Over the years, the wealth of knowledge accumulated as design principles has reached a critical mass. Entire books are now dedicated to the subject. Still, despite this body of knowledge, design remains difficult. A major problem is that principles are ambiguous and not very constructive. Ambiguity comes from the use of natural language, but also from the desire to capture a large number of cases under a single principle. As a consequence, adherence to a principle is often nonmeasurable.

By "not very constructive," I mean that a principle often provides no guidance toward a design that respects the principle itself. This makes principles-driven design somewhat hard to practice successfully unless you have considerable experience.

Patterns are a great way to encode knowledge, but as a design technique they are incomplete: they leverage existing solutions but cannot generate new ones.

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

THE REVENGE OF PRACTICE

In more recent times, design patterns have emerged as a valid alternative to the principles-driven approach. The introduction to *Design Patterns* (E. Gamma et al., Addison-Wesley, 1994) is adamant: "One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past."

If principles represent the "say-what" approach to design, patterns are the "show-how" way. You look for recurring design problems, find the solutions that worked better, and capture their essence in a pattern.

The biggest problem with patterns is that they rely on previous experience, so when the available knowledge does not cover your problem, you are out of luck. Sure, human beings are fairly flexible; they can come up with some workable solution anyway. But obviously there is a limit to how far you can go with patterns alone. Also, patterns leave a lot of details unspecified, so that they are independent from each instance. As a consequence, designers must be able to fill in the blank, usually backing up with their own experience, which is hard for inexperienced designers.

Finally, there is a "disregard for originality" in the pattern movement. Stop and think about how those clever solutions have been devised in the first place. Because in absence of previous experience, the original, bright designer didn't use patterns, but some other approach which is now intentionally neglected. Patterns are a great way to encode knowledge, but as a design technique they are necessarily incomplete: They leverage existing solutions but cannot generate completely new ones.

SYSTEMATIC DESIGN

It is recognized that we lack a systematic approach to design. Assuming we have done a good job of analysis, we are still largely left at the mercy of experience: our own experience, to disentangle ourselves in the jungle of principles, and other people's experience, under the form of patterns. This being the same in all the other disciplines, we may conclude that it is an unfortunate fact of life, shrug our shoulders, and go back to our high-paying design jobs. But this is not generally true. Electrical engineering, for instance, has a large theoretical background but also a comprehensive body of heuristic techniques that can be systematically applied to solve problems. The same holds for other fields as well. And here we close the circle: the Feigenbaum Bottleneck notwithstanding, we have to bite the bullet and try to articulate a "say-how" approach to software design.

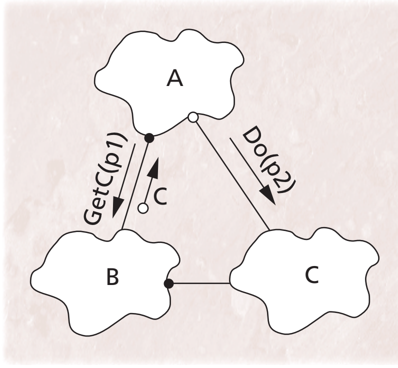

Figure 1. Violating a principle.

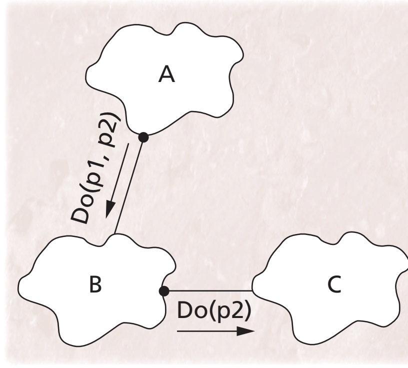

Figure 2. Sample transformation.

TRANSFORMATIONAL APPROACH

Trivial design is easy. After all, it is just a matter of laying down a solution, and most people can even skip this step altogether and jump to coding. Each approach leads to a solution that is often less than optimal.

However, if we have a systematic method to recognize and solve common problems in the initial, trivial design, we have a much higher chance of ending up with a better result. Also, if the problem-solving approach is constructive, we can gradually learn how to avoid problems up front, applying the techniques on the fly as expert designers do.

Here I describe an approach to design based on transformation techniques, which I've used with success in the development of large systems. To find a transformation technique, you start with an informal design principle. Then you try to distillate some necessary conditions which are unambiguous and amenable to formal verification. It is not necessary to capture the whole essence of the principle in a single rule: In some cases, that's simply not possible. However, we can settle for a small number of necessary conditions, provided that they can be conceptually ascribed to the principle and that they are formally verifiable. The next step is to identify a set of transformations that can be used to enforce the observance of a rule.

Each transformation technique, when applied to an interoperating set of classes that violate a design rule, should lead to an equivalent structure that respects the rule. Patterns are a good source of inspiration for atomic transformation techniques because most patterns are based on a combination of several simpler techniques. That's why patterns aren't always easy to understand in depth.

Consider the informal principle, "abstractions should not depend on details." I have taken this principle and identified a formal necessary condition and some transformation techniques (see my paper "Deriving Patterns from Design Principles," to appear this year in the *Journal of Object Oriented Programming*). Using those techniques, I derive the Observer and the Mediator patterns from two trivial designs.

Here I briefly consider a simpler case, the "I'm Alive!" principle, as described by Peter Coad and Jill Nicola (*Object Oriented Programming*, Yourdon Press, 1993). This principle is usually spoken as "an object should expose services, not data." This is a fundamental OOP principle, but unfortunately it is not based on measurable semantics. In practice, it can be violated as follows: Object A asks object B for one of its parts, C, and then does something with C. Figure 1 shows an example.

For example, B may be a Book, C a Chapter, and A may iterate over the chapters and print them. If we consider only this scheme, it's much easier to come up with a good design rule that can be automatically verified. In fact, such a rule already exists. It's called the Demeter Law: Any method M of a class C should use only the immediate parts of the object, the parameters of M, objects that are directly created in M, or global objects (described by K.J. Lieberherr and I.M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Sept. 1989). Other formulations are more amenable to automatic checking.

Basically, the Demeter Law prevents an object from taking a part of another object and doing something with it. Any program can be modified to conform with the Demeter Law, but automatic transformations rarely yield good code. What we want is to find a set of transformations that respect the spirit of the rules, not only the letter. Our job is to find the verifiable condition, and to provide a set of transformations to the designer. The designer picks the best suited transformation.

Figure 2 gives an example of a transformation. Now A asks B to do something, and B gets its own part and asks it to do it. Instead of asking for a chapter and then printing it, now A asks to print a certain chapter. Each object exposes only services, not data (the "get" becomes "do").

Although trivial once exposed, this transformation rule is useful in a large number of cases and is used instinctively by experienced designers. Naturally, there are several other transformation techniques that may be used to enforce the Demeter Law, as well as other measurable, necessary conditions that subsume the "I'm Alive!" principle. If you know some interesting techniques, I would be glad to hear from you.

Over the years, I've identified a small set of verifiable rules and transformation techniques. I've applied them during the design of large, real-world critical systems, and taught them to developers. I've found that they are very easy to grasp and that people end up paying more attention to design principles once they know how to verify and make correction to their works. Obviously, many rules and techniques are yet to be found. Interestingly, verifiable rules may lead to some true object-oriented CASE tools, beyond glorified drawing systems. ❖

***Carlo Pescio** is a consultant and mentor for various European companies and corporations, specializing in object-oriented technologies. He has a doctoral degree in computer science and is a member of the IEEE Computer Society, the ACM, and the New York Academy of Sciences. Contact him at pescio@acm.org.*