Editor: Martin Fowler, Thought Works ■ fowler@acm.org

Protected Variation: The Importance of Being Closed

Craig Larman

The *Pattern Almanac 2000* (Addison-Wesley, 2000) lists around 500 software-related patterns—and given this reading list, the curious developer has no time to program! Of course, there are underlying, simplifying themes and principles to this pattern plethora that developers have long considered and discussed. One example is Larry Constantine's coupling and cohesion guidelines (see "Structured Design," *IBM Systems J.*, vol. 13, no. 2, 1974). Yet, these principles must continually resurface to help each new generation of developers and architects cut through the apparent disparity in myriad design ideas and help them see the underlying and unifying forces.

One such principle, which Bertrand Meyer describes in *Object-Oriented Software Construction* (IEEE Press, 1988), is the Open–Closed Principle: *Modules should be both open (for extension and adaptation) and closed (to avoid modification that affect clients)*. OCP is essentially equivalent to the Protected Variation pattern: *Identify points of predicted variation and create a stable interface around them.* Alistair Cockburn did not know of OCP when he first wrote about PV (see "Prioritizing Forces in Software Design," *Patterns Languages of Program Design*, vol. 2, Addison-Wesley, 1996). Furthermore, OCP is what David Parnas really meant by *information hiding* (see "On the Criteria to Be Used in Decomposing Systems into Modules" *Comm. ACM*, vol. 12, no. 2, Dec. 1972).

OCP and PV formalize and generalize a common and fundamental design principle described in many guises. OCP and PV are two expressions of the same principle—protection against change to the existing code and design at variation and evolution points—with minor differences in emphasis. I am nominating the term *protected variation* for general use, as it is short and clear.

In OCP, the term *module* includes all discrete software elements, including methods, classes, subsystems, applications, and so forth. Also, the phrase "closed with respect to X" means that clients are not affected if X changes. For example, "The class is closed with respect to instance field definitions." PV uses the term *interface* in the broad sense of an access view—not exactly a Java or COM interface, for example.

Information hiding is PV, not data encapsulation

"On the Criteria To Be Used in Decomposing Systems Into Modules" is a classic that is often cited but seldom read. In it, Parnas introduces information hiding. Many people have misinterpreted the term as meaning data encapsulation, and some books erroneously define the concepts as synonyms.

Parnas intended it to mean *hide information about the design from other modules, at the points of difficult or likely change*. To quote his discussion of information hiding as a guiding design principle:

> *We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each*

0740-7459/01/$10.00 © 2001 IEEE

module is then designed to hide such a decision from the others.

That is, Parnas's information hiding is the same principle expressed in PV or OCP—it is not simply data encapsulation, which is but one of many techniques to hide design information.

However, the term has been so widely reinterpreted as a synonym for data encapsulation that it is no longer possible to use it in its original sense without misunderstanding it. This article should be called, "The Importance of Information Hiding," in honor of Parnas's description of the PV principle. Dijkstra earlier alludes to the principle in the "THE" project, but Parnas gave it focus and shape (Dijkstra, "The Structure of the 'THE' Multiprogramming System," *Comm. ACM*, 1968).

Mechanisms motivated by PV

PV is a root principle motivating most of the mechanisms and patterns in programming and design that provide flexibility and protection from variations. Here are some examples.

Familiar PV mechanisms

PV motivates data encapsulation, interfaces, polymorphism, indirection, and standards. Components such as brokers and virtual machines are complex examples of indirection.

Uniform access

Languages such as Ada, Eiffel, and C# support a syntactic construct to express both a method and field access in the same way. For example, *aCircle.radius* might invoke a *radius():float* method or directly refer to a public field, depending on the definition of the class. You can change public fields to access methods without changing the client code.

Data-driven designs

Data-driven designs cover a broad family of techniques, including reading codes, values, class file paths, class names, and so forth, from an external source in order to change the behavior of or "parameterize" a system in some way at runtime. Other variants include style sheets, metadata for object-relational mapping, property files, reading in window layouts, and much more. The system is protected from the impact of data, metadata, or declarative variations by externalizing the variant, reading the behavior-influencing data in, and reasoning with it.

Service lookup

Service lookup includes techniques such as using naming services (for example, JNDI) or traders to obtain a service (such as Jini). This approach uses the stable interface of the lookup service to protect clients from variations in the location of services. It is a special case of data-driven designs.

Interpreter-driven designs

Interpreter-driven designs include rule interpreters that execute rules read from an external source, script or language interpreters that read and run programs, virtual machines, neural network engines that execute nets, constraint logic engines that read and reason with constraint sets, and so forth. This approach lets you change or parameterize a system's behavior through external logic expressions. The system is protected from the impact of logic variations by externalizing the logic, reading it in (for example, rules or a neural net), and using an interpreter.

Reflective or metalevel designs

An example of a reflective or metalevel design includes using the java.beans.Introspector to obtain a BeanInfo object, asking for the getter Method object for bean property X (that is, the method *getX*), and calling Method.invoke. Reflective algorithms that use introspection and metalanguage services protect the system from the impact of logic or external code variations. We could also consider this a special case of data-driven designs.

You must pick your battles in design, be they at the macro-architectural level or the humble instance field.

Pick your battles

As an example of PV's application, a client explained that the logistical support application used by an airline was a maintenance headache. There was frequent modification of the business logic to support the logistics. How do you protect the system from variations at this point? From the mechanisms to support PV (data encapsulation, interfaces, indirection, …), a rule-based design was chosen: A rules engine was added to the system, and an external rule editor let the subject matter experts update the rules without requiring changes to the source code of the system.

Low coupling and protection against variations is not motivated in all areas. You must pick your battles in design, be they at the macro-architectural level or the humble instance field. A good designer can identify the likely points of instability or variation and apply PV to those points but not others. Otherwise, effort is wasted and complexity may arise (and with it, the chance for defects).

For example, I recall being surprised by the occasional use of *static public final* fields in the Java technology libraries (after spending many years with the Smalltalk libraries). Some might be poorly conceived, but some, such as the *Color* static fields *red*, *black*, *white*, and so forth, are extremely stable; the likelihood of instability is so low that making them private and adding accessing methods is just object purism.

As a counterexample, I know of a pager-message-handling system in which the architect added a fancy scripting language and interpreter to support some flexibility. However, during rework in an incremental release, the complex (and inefficient) scripting was removed—it wasn't needed.

Judicious PV and the Diamond Sutra

Constantine's guideline to design with low coupling is a truly core prin-

ciple of design, and it can be argued that PV derives from it. We can prioritize our goals and strategies as follows:

1. We wish to save time and money, reduce the introduction of new defects, and reduce the pain and suffering inflicted on overworked developers.
2. To achieve this, we design to minimize the impact of change.
3. To minimize change impact, we design with the goal of low coupling.
4. To design for low coupling, we design for PVs.

Low coupling and PV are just one set of mechanisms to achieve the goals of saving time, money, and so forth. Sometimes, the cost of speculative future proofing to achieve these goals outweighs the cost incurred by a simple, highly coupled "brittle" design that is reworked as necessary in response to true change pressures. That is, the cost of engineering protection at evolution points can be higher than reworking a simple design.

My point is not to advocate rework and brittle designs. If the need for flexibility and PV is immediately applicable, then applying PV is justified.

However, if you're using PV for speculative future proofing or reuse, then deciding which strategy to use is not as clear-cut. Novice developers tend toward brittle designs, and intermediates tend toward overly fancy and flexible generalized ones (in ways that never get used). Experts choose with insight—perhaps choosing a simple and brittle design whose cost of change is balanced against its likelihood. The journey is analogous to the well-known stanza from the Diamond Sutra:

Before practicing Zen, mountains were mountains and rivers were rivers.

While practicing Zen, mountains are no longer mountains and rivers are no longer rivers.

After realization, mountains are mountains and rivers are rivers again.

PV is a fundamental design principle that applies to everything from the largest architectural concerns to the smallest coding decision. Furthermore, it underlies the motivation and advice of most other patterns and principles. As Parnas explained 30 years ago—and as has resurfaced in the writings of Meyer and Cockburn—each generation of software developers needs help seeing mountains as mountains again—especially after four years of computer science and 500 patterns!

Acknowledgments

OCP, as described here, was brought to my attention by Bob Martin in *The Open-Closed Principle: C++ Report*, SIGS Publications, 1996.

Craig Larman is director of process and methodology at Valtech, an international consulting group. He holds a BSc and an MSc in computer science, with research emphasis in artificial intelligence. He is the author of *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, and he is writing a second edition that will include OCP/PV as one of the fundamental design principles. He is a member of the IEEE and ACM. Contact him at clarman@acm.org.

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@ computer.org), or access computer.org/software/author.htm.

Letters to the Editor
Send letters to

Letters Editor
IEEE Software
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access computer.org/software for information about *IEEE Software*.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address
Send change-of-address requests for the membership directory to help@computer.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.