The Open Closed Principle

Uncle Bob → 12 May 2014 + Craftsmanship

Share Tweet Share

In 1988 Bertrand Meyer defined one of the most important principles of software engineering. The Open Closed Principle (OCP). In his book Object Oriented Software Construction[1] he said:

> A satisfactory modular decomposition technique must satisfy one more requirement: It should yield modules that are *both* open and closed.
>
> - A module will be said to be open if it is available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
> - A module will be said to be closed if is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding). In the case of a programming language module, a closed module is one that may be compiled and stored in a library, for others to use. In the case of a design or specification

Kevin Buchanan

Lihsuan Lung

Eric Meyer
Malcolm Newsome
Myles Megyesi

Chris Peak
Dave Moore

Mike Knepper

Margaret Pagel

Steve Kim
Kevin Liddle

Brian Pratt
Aaron Lahey

Christoph Gockel

Kofi Gumbs

Chris Jordan

Connor Mendenhall

Makis Otman

Mike Ebert
Georgina Mcfadyen

> module, closing a module simply means having it approved by management, adding it to the project's official repository of accepted software items (often called the project *baseline*), and publishing its interface for the benefit of other module designers.

This definition is obviously dated. Nowadays many languages don't require that modules be compiled. And getting module specifications approved by management smacks of a waterfall mentality. Still, the essence of a great principle still shines through those old words. To wit:

> You should be able to extend the behavior of a system *without having to modify that system.*

Think about that very carefully. If the behaviors of all the modules in your system could be extended, without modifying them, then you could add new features to that system *without modifying any old code*. The features would be added solely by writing new code.

What's more, since none of the old code had changed, it would not need to be recompiled, and therefore it would not need to be redeployed. Adding a new feature would involve leaving the old code in place and *only deploying the new code*, perhaps in a new `jar` or `dll` or `gem`.

And this ought to give you a hint about what a `jar`, `dll`, or `gem` really ought to be. They ought to be isolatable *features*!

Is this Absurd?

At first reading the open closed principle may seem to be nonsensical. Our languages and our designs do not usually allow new features to be written, compiled, and deployed separately from the rest of the system. We seldom find ourselves in a place where the current system is closed for modification, and yet can be extended with new features.

Indeed, most commonly we add new features by making a vast number of changes throughout the body of the existing code. We've known this was bad long before Martin Fowler wrote the book[2] that labeled it *Shotgun Surgery*; but we still do it.

Ah, but then there's Eclipse, or IntelliJ, or Visual Studio, or Vim, or Text Mate, or Minecraft or... Well, you get my point. There is a vast plethora of tools that can be easily extended without modifying or redeploying them. We extend them by writing *plugins*.

Plugin systems are the ultimate consummation, the apotheosis, of the Open-Closed Principle. They are proof positive that open-closed systems are possible, useful, and immensely powerful.

How did these systems manage to close their primary business rules to modification, and yet leave the whole application open to be extended? Simple. They believed in the OCP, and they used the tools of object oriented design to separate high level policy from low level detail. They carefully managed their dependencies, inverting those that crossed architecturally significant boundaries in the wrong direction.

After all, the way you get a plugin architecture is to make sure that all dependencies inside the plugin, point at the system; and that nothing in the system points out towards the plugins. The system doesn't know about the plugins. The plugins know about the system.

Plugin Architecture

What if the design of your systems was based around plugins, like Vim, Emacs, Minecraft, or Eclipse? What if you could plug in the Database, or the GUI. What if you could plug in new features, and unplug old ones. What if the behavior of your system was largely controlled by the configuration of its plugins? What power would that give you? How easy would it be to add new features, or new user interfaces, or new machine/machine interfaces? How easy would it be to add, or remove, SOA. How easy would it be to add or remove REST? How easy would it be to add or remove Spring, or Rails, or Hibernate, or Oracle, or...

Well I think you get my meaning. When your fundamental business rules are the core of a plugin architecture, then you are never bound to a particular feature set, interface, database, framework, or anything else.

Conclusion

I've heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be *the most important aspect* of future software systems.

---

[1] *Object Oriented Software Construction*, 1st. ed. Bertrand Meyer, Prentice Hall, 1988. [2] *Refactoring*, Martin Fowler, Addison Wesley, 1999

Share Tweet G+ Share

Robert Martin (Uncle Bob) is a Master Craftsman. He's an award-winning author, renowned speaker, and has been an über software geek since 1970.

Follow @unclebobmartin

RELATED POSTS

Make Yourself Dispensable Lihsuan Lung
The Third Bricklayer Robert Gu
The egoless programmer Daniel Irvine
Fundamental tensions in code smells Colin Jones
A Good Craftsman Never Blames His Tools Georgina Mcfadyen
Are you a wise developer? Malcolm Newsome
8th Light Acquires WisdomGroup Paul Pagel
The Onboarding Checklist Lihsuan Lung
What will your legacy [code] be? Colin Jones
Refactor and Source-Control Your CI Dariusz Pasciak

MORE POSTS BY THIS AUTHOR

Future Proof
Agile is not now, nor was it ever, Waterfall.
VW
WATS Line 54
A Little Structure
Make the Magic go away.
The Little Singleton
The First Micro-service Architecture
Language Layers
Does organization matter?

Interested in 8th Light's services? Let's talk.

Contact Us