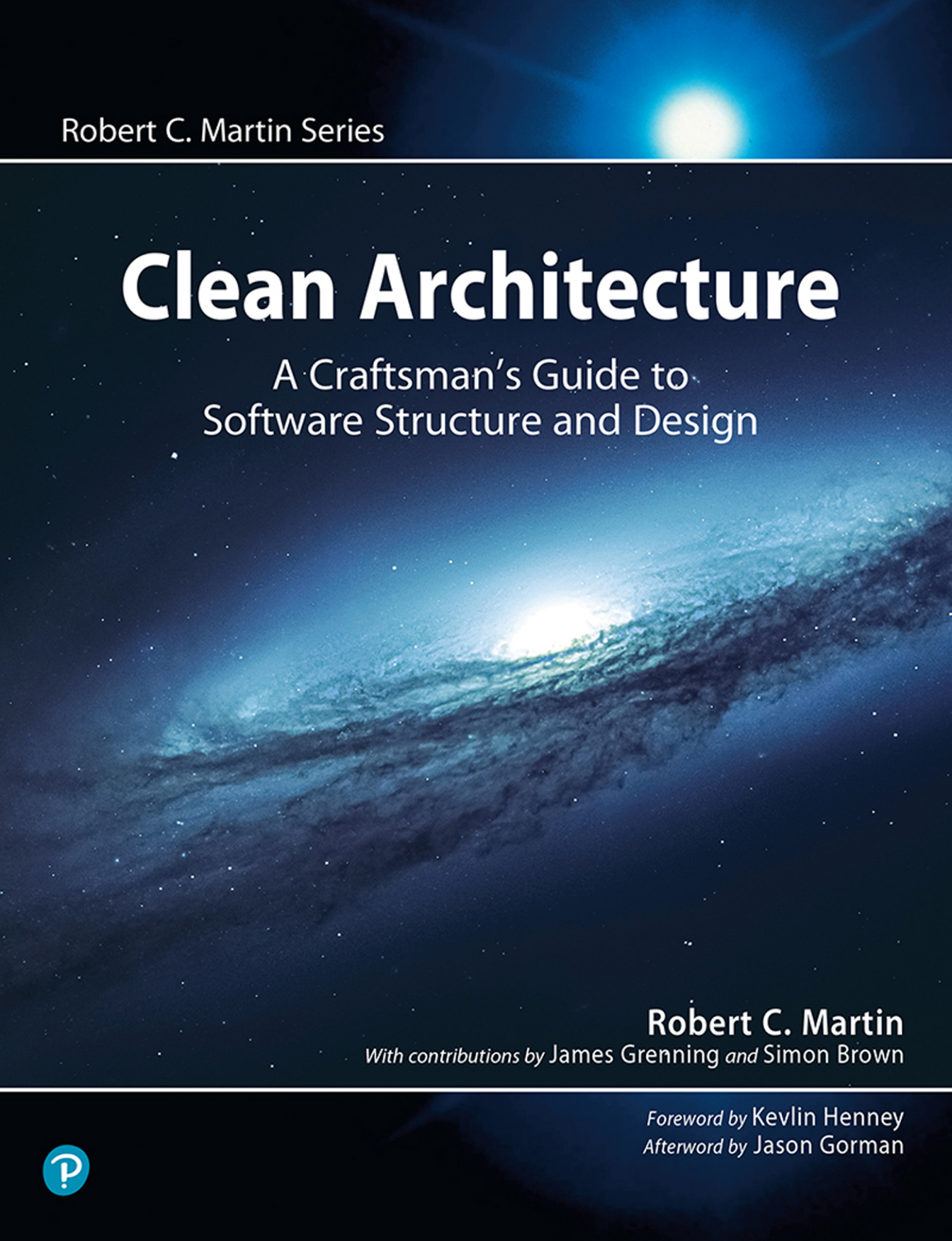
Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to
Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman

Clean Architecture

allowed programmers to prepare an executable using the slow linker, but then they could load it quickly, at any time.

Then came the 1980s. Programmers were working in C or some other high-level language. As their ambitions grew, so did their programs. Programs that numbered hundreds of thousands of lines of code were not unusual.

Source modules were compiled from `.c` files into `.o` files, and then fed into the linker to create executable files that could be quickly loaded. Compiling each individual module was relatively fast, but compiling *all* the modules took a bit of time. The linker would then take even more time. Turnaround had again grown to an hour or more in many cases.

It seemed as if programmers were doomed to endlessly chase their tails. Throughout the 1960s, 1970s, and 1980s, all the changes made to speed up workflow were thwarted by programmers' ambitions, and the size of the programs they wrote. They could not seem to escape from the hour-long turnaround times. Loading time remained fast, but compile-link times were the bottleneck.

We were, of course, experiencing Murphy's law of program size:

> *Programs will grow to fill all available compile and link time.*

But Murphy was not the only contender in town. Along came Moore,[3] and in the late 1980s, the two battled it out. Moore won that battle. Disks started to shrink and got significantly faster. Computer memory started to get so ridiculously cheap that much of the data on disk could be cached in RAM. Computer clock rates increased from 1 MHz to 100 MHz.

By the mid-1990s, the time spent linking had begun to shrink faster than our ambitions could make programs grow. In many cases, link time decreased to a matter of *seconds*. For small jobs, the idea of a linking loader became feasible again.

3. Moore's law: Computer speed, memory, and density double every 18 months. This law held from the 1950s to 2000, but then, at least for clock rates, stopped cold.

This was the era of Active-X, shared libraries, and the beginnings of `.jar` files. Computers and devices had gotten so fast that we could, once again, do the linking at load time. We could link together several `.jar` files, or several shared libraries in a matter of seconds, and execute the resulting program. And so the component plugin architecture was born.

Today we routinely ship `.jar` files or DLLs or shared libraries as plugins to existing applications. If you want to create a mod to *Minecraft*, for example, you simply include your custom `.jar` files in a certain folder. If you want to plug *Resharper* into *Visual Studio*, you simply include the appropriate DLLs.

CONCLUSION

These dynamically linked files, which can be plugged together at runtime, are the software components of our architectures. It has taken 50 years, but we have arrived at a place where component plugin architecture can be the casual default as opposed to the herculean effort it once was.

13 Component Cohesion

Which classes belong in which components? This is an important decision, and requires guidance from good software engineering principles. Unfortunately, over the years, this decision has been made in an ad hoc manner based almost entirely on context.

In this chapter we will discuss the three principles of component cohesion:

- **REP:** The Reuse/Release Equivalence Principle
- **CCP:** The Common Closure Principle
- **CRP:** The Common Reuse Principle

THE REUSE/RELEASE EQUIVALENCE PRINCIPLE

The granule of reuse is the granule of release.

The last decade has seen the rise of a menagerie of module management tools, such as Maven, Leiningen, and RVM. These tools have grown in importance because, during that time, a vast number of reusable components and component libraries have been created. We are now living in the age of software reuse—a fulfillment of one of the oldest promises of the object-oriented model.

The Reuse/Release Equivalence Principle (REP) is a principle that seems obvious, at least in hindsight. People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.

This is not simply because, without release numbers, there would be no way to ensure that all the reused components are compatible with each other. Rather, it also reflects the fact that software developers need to know when new releases are coming, and which changes those new releases will bring.

It is not uncommon for developers to be alerted about a new release and decide, based on the changes made in that release, to continue to use the old

release instead. Therefore the release process must produce the appropriate notifications and release documentation so that users can make informed decisions about when and whether to integrate the new release.

From a software design and architecture point of view, this principle means that the classes and modules that are formed into a component must belong to a cohesive group. The component cannot simply consist of a random hodgepodge of classes and modules; instead, there must be some overarching theme or purpose that those modules all share.

Of course, this should be obvious. However, there is another way to look at this issue that is perhaps not quite so obvious. Classes and modules that are grouped together into a component should be *releasable* together. The fact that they share the same version number and the same release tracking, and are included under the same release documentation, should make sense both to the author and to the users.

This is weak advice: Saying that something should "make sense" is just a way of waving your hands in the air and trying to sound authoritative. The advice is weak because it is hard to precisely explain the glue that holds the classes and modules together into a single component. Weak though the advice may be, the principle itself is important, because violations are easy to detect—they don't "make sense." If you violate the REP, your users will know, and they won't be impressed with your architectural skills.

The weakness of this principle is more than compensated for by the strength of the next two principles. Indeed, the CCP and the CRP strongly define the this principle, but in a negative sense.

THE COMMON CLOSURE PRINCIPLE

Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change at different times and for different reasons.

This is the Single Responsibility Principle restated for components. Just as the SRP says that a *class* should not contain multiples reasons to change, so the Common Closure Principle (CCP) says that a *component* should not have multiple reasons to change.

For most applications, maintainability is more important than reusability. If the code in an application must change, you would rather that all of the changes occur in one component, rather than being distributed across many components.[1] If changes are confined to a single component, then we need to redeploy only the one changed component. Other components that don't depend on the changed component do not need to be revalidated or redeployed.

The CCP prompts us to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same component. This minimizes the workload related to releasing, revalidating, and redeploying the software.

This principle is closely associated with the Open Closed Principle (OCP). Indeed, it is "closure" in the OCP sense of the word that the CCP addresses. The OCP states that classes should be closed for modification but open for extension. Because 100% closure is not attainable, closure must be strategic. We design our classes such that they are closed to the most common kinds of changes that we expect or have experienced.

The CCP amplifies this lesson by gathering together into the same component those classes that are closed to the same types of changes. Thus, when a change in requirements comes along, that change has a good chance of being restricted to a minimal number of components.

1. See the section on "The Kitty Problem" in Chapter 27, "Services: Great and Small."