Clean Architecture

A Craftsman's Guide to Software Structure and Design

Robert C. Martin

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com

Library of Congress Control Number: 2017945537

ISBN-13: 978-0-13-449416-6
ISBN-10: 0-13-449416-4

1 17

This book is dedicated to my lovely wife, my four spectacular children, and their families, including my quiver full of five grandchildren—who are the dessert of my life.

This page intentionally left blank

Contents

Foreword

What do we talk about when we talk about architecture?

As with any metaphor, describing software through the lens of architecture can hide as much as it can reveal. It can both promise more than it can deliver and deliver more than it promises.

The obvious appeal of architecture is structure, and structure is something that dominates the paradigms and discussions of software development—components, classes, functions, modules, layers, and services, micro or macro. But the gross structure of so many software systems often defies either belief or understanding—Enterprise Soviet schemes destined for legacy, improbable Jenga towers reaching toward the cloud, archaeological layers buried in a big-ball-of-mud slide. It's not obvious that software structure obeys our intuition the way building structure does.

Buildings have an obvious physical structure, whether rooted in stone or concrete, whether arching high or sprawling wide, whether large or small, whether magnificent or mundane. Their structures have little choice but to respect the physics of gravity and their materials. On the other hand—except in its sense of seriousness—software has little time for gravity. And what is software made of? Unlike buildings, which may be made of bricks, concrete,

wood, steel, and glass, software is made of software. Large software constructs are made from smaller software components, which are in turn made of smaller software components still, and so on. It's coding turtles all the way down.

When we talk about software architecture, software is recursive and fractal in nature, etched and sketched in code. Everything is details. Interlocking levels of detail also contribute to a building's architecture, but it doesn't make sense to talk about physical scale in software. Software has structure—many structures and many kinds of structures—but its variety eclipses the range of physical structure found in buildings. You can even argue quite convincingly that there is more design activity and focus in software than in building architecture—in this sense, it's not unreasonable to consider software architecture more architectural than building architecture!

But physical scale is something humans understand and look for in the world. Although appealing and visually obvious, the boxes on a PowerPoint diagram are not a software system's architecture. There's no doubt they represent a particular view of an architecture, but to mistake boxes for *the* big picture—for *the* architecture—is to miss the big picture and the architecture: Software architecture doesn't look like anything. A particular visualization is a choice, not a given. It is a choice founded on a further set of choices: what to include; what to exclude; what to emphasize by shape or color; what to de-emphasize through uniformity or omission. There is nothing natural or intrinsic about one view over another.

Although it might not make sense to talk about physics and physical scale in software architecture, we do appreciate and care about certain physical constraints. Processor speed and network bandwidth can deliver a harsh verdict on a system's performance. Memory and storage can limit the ambitions of any code base. Software may be such stuff as dreams are made on, but it runs in the physical world.

> *This is the monstrosity in love, lady, that the will is infinite, and the execution confined; that the desire is boundless, and the act a slave to limit.*
>
> —William Shakespeare

The physical world is where we and our companies and our economies live. This gives us another calibration we can understand software architecture by, other less physical forces and quantities through which we can talk and reason.

> *Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.*
>
> —Grady Booch

Time, money, and effort give us a sense of scale to sort between the large and the small, to distinguish the architectural stuff from the rest. This measure also tells us how we can determine whether an architecture is good or not: Not only does a good architecture meet the needs of its users, developers, and owners at a given point in time, but it also meets them over time.

> *If you think good architecture is expensive, try bad architecture.*
>
> —Brian Foote and Joseph Yoder

The kinds of changes a system's development typically experiences should not be the changes that are costly, that are hard to make, that take managed projects of their own rather than being folded into the daily and weekly flow of work.

That point leads us to a not-so-small physics-related problem: time travel. How do we know what those typical changes will be so that we can shape those significant decisions around them? How do we reduce future development effort and cost without crystal balls and time machines?

> *Architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other.*
>
> —Ralph Johnson

Understanding the past is hard enough as it is; our grasp of the present is slippery at best; predicting the future is nontrivial.

This is where the road forks many ways.

Down the darkest path comes the idea that strong and stable architecture comes from authority and rigidity. If change is expensive, change is eliminated—its causes subdued or headed off into a bureaucratic ditch. The architect's mandate is total and totalitarian, with the architecture becoming a dystopia for its developers and a constant source of frustration for all.

Down another path comes a strong smell of speculative generality. A route filled with hard-coded guesswork, countless parameters, tombs of dead code, and more accidental complexity than you can shake a maintenance budget at.

The path we are most interested is the cleanest one. It recognizes the softness of software and aims to preserve it as a first-class property of the system. It recognizes that we operate with incomplete knowledge, but it also understands that, as humans, operating with incomplete knowledge is something we do, something we're good at. It plays more to our strengths than to our weaknesses. We create things and we discover things. We ask questions and we run experiments. A good architecture comes from understanding it more as a journey than as a destination, more as an ongoing process of enquiry than as a frozen artifact.

> *Architecture is a hypothesis, that needs to be proven by implementation and measurement.*
>
> —Tom Gilb

To walk this path requires care and attention, thought and observation, practice and principle. This might at first sound slow, but it's all in the way that you walk.

> *The only way to go fast, is to go well.*
>
> —Robert C. Martin

Enjoy the journey.

—Kevlin Henney
May 2017

Preface

The title of this book is *Clean Architecture*. That's an audacious name. Some would even call it arrogant. So why did I choose that title, and why did I write this book?

I wrote my very first line of code in 1964, at the age of 12. The year is now 2016, so I have been writing code for more than half a century. In that time, I have learned a few things about how to structure software systems—things that I believe others would likely find valuable.

I learned these things by building many systems, both large and small. I have built small embedded systems and large batch processing systems. I have built real-time systems and web systems. I have built console apps, GUI apps, process control apps, games, accounting systems, telecommunications systems, design tools, drawing apps, and many, many others.

I have built single-threaded apps, multithreaded apps, apps with few heavy-weight processes, apps with many light-weight processes, multiprocessor apps, database apps, mathematical apps, computational geometry apps, and many, many others.

I've built a lot of apps. I've built a lot of systems. And from them all, and by taking them all into consideration, I've learned something startling.

The architecture rules are the same!

This is startling because the systems that I have built have all been so radically different. Why should such different systems all share similar rules of architecture? My conclusion is that *the rules of software architecture are independent of every other variable*.

This is even more startling when you consider the change that has taken place in hardware over the same half-century. I started programming on machines the size of kitchen refrigerators that had half-megahertz cycle times, 4K of core memory, 32K of disk memory, and a 10 character per second teletype interface. I am writing this preface on a bus while touring in South Africa. I am using a MacBook with four i7 cores running at 2.8 gigahertz each. It has 16 gigabytes of RAM, a terabyte of SSD, and a 2880×1800 retina display capable of showing extremely high-definition video. The difference in computational power is staggering. Any reasonable analysis will show that this MacBook is at least 10^{22} more powerful than those early computers that I started using half a century ago.

Twenty-two orders of magnitude is a very large number. It is the number of angstroms from Earth to Alpha-Centuri. It is the number of electrons in the change in your pocket or purse. And yet that number—that number *at least*—is the computational power increase that I have experienced in my own lifetime.

And with all that vast change in computational power, what has been the effect on the software I write? It's gotten bigger certainly. I used to think 2000 lines was a big program. After all, it was a full box of cards that weighed 10 pounds. Now, however, a program isn't really big until it exceeds 100,000 lines.

The software has also gotten much more performant. We can do things today that we could scarcely dream about in the 1960s. *The Forbin Project*, *The*

Moon Is a Harsh Mistress, and *2001: A Space Odyssey* all tried to imagine our current future, but missed the mark rather significantly. They all imagined huge machines that gained sentience. What we have instead are impossibly small machines that are still … just machines.

And there is one thing more about the software we have now, compared to the software from back then: *It's made of the same stuff.* It's made of `if` statements, assignment statements, and `while` loops.

Oh, you might object and say that we've got much better languages and superior paradigms. After all, we program in Java, or C#, or Ruby, and we use object-oriented design. True—and yet the code is still just an assemblage of sequence, selection, and iteration, just as it was back in the 1960s and 1950s.

When you really look closely at the practice of programming computers, you realize that very little has changed in 50 years. The languages have gotten a little better. The tools have gotten fantastically better. But the basic building blocks of a computer program have not changed.

If I took a computer programmer from 1966 forward in time to 2016 and put her[1] in front of my MacBook running IntelliJ and showed her Java, she might need 24 hours to recover from the shock. But then she would be able to write the code. Java just isn't that different from C, or even from Fortran.

And if I transported you back to 1966 and showed you how to write and edit PDP-8 code by punching paper tape on a 10 character per second teletype, you might need 24 hours to recover from the disappointment. But then you would be able to write the code. The code just hasn't changed that much.

That's the secret: This changelessness of the code is the reason that the rules of software architecture are so consistent across system types. The rules of software architecture are the rules of ordering and assembling the building

1. And she very likely would be female since, back then, women made up a large fraction of programmers.

blocks of programs. And since those building blocks are universal and haven't changed, the rules for ordering them are likewise universal and changeless.

Younger programmers might think this is nonsense. They might insist that everything is new and different nowadays, that the rules of the past are past and gone. If that is what they think, they are sadly mistaken. The rules have not changed. Despite all the new languages, and all the new frameworks, and all the paradigms, the rules are the same now as they were when Alan Turing wrote the first machine code in 1946.

But one thing has changed: Back then, we didn't know what the rules were. Consequently, we broke them, over and over again. Now, with half a century of experience behind us, we have a grasp of those rules.

And it is those rules—those timeless, changeless, rules—that this book is all about.

Register your copy of *Clean Architecture* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134494166) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access the bonus materials.

Acknowledgments

The people who played a part in the creation of this book—in no particular order:

Chris Guzikowski
Chris Zahn
Matt Heuser
Jeff Overbey
Micah Martin
Justin Martin
Carl Hickman
James Grenning
Simon Brown
Kevlin Henney
Jason Gorman
Doug Bradbury
Colin Jones
Grady Booch
Kent Beck

Martin Fowler
Alistair Cockburn
James O. Coplien
Tim Conrad
Richard Lloyd
Ken Finder
Kris Iyer (CK)
Mike Carew
Jerry Fitzpatrick
Jim Newkirk
Ed Thelen
Joe Mabel
Bill Degnan

And many others too numerous to name.

In my final review of this book, as I was reading the chapter on Screaming Architecture, Jim Weirich's bright-eyed smile and melodic laugh echoed through my mind. Godspeed, Jim!

About the Author

Robert C. Martin (Uncle Bob) has been a programmer since 1970. He is the co-founder of cleancoders.com, which offers online video training for software developers, and is the founder of Uncle Bob Consulting LLC, which offers software consulting, training, and skill development services to major corporations worldwide. He served as the Master Craftsman at 8th Light, Inc., a Chicago-based software consulting firm. He has published dozens of articles in various trade journals and is a regular speaker at international conferences and trade shows. He served three years as the editor-in-chief of the *C++ Report* and served as the first chairman of the Agile Alliance.

Martin has authored and edited many books, including *The Clean Coder*, *Clean Code*, *UML for Java Programmers*, *Agile Software Development*, *Extreme Programming in Practice*, *More C++ Gems*, *Pattern Languages of Program Design 3*, and *Designing Object Oriented C++ Applications Using the Booch Method*.