I don’t usually read books on software engineering—these days, they all pretty much say the same thing. However, A Handbook of Software and Systems Engineering offers something unique. Albert Endres and Dieter Rombach provide the much needed scientific basis for software engineering. Furthermore, for the first time, they bring together all the rules, laws, and theories of software and systems engineering, presented in a unified fashion and grounded in empirical observations. Well-researched and nicely presented, the book should be of tremendous value to serious software engineering professionals and researchers alike.

Presenting a theoretical foundation

Structuring their treatment along the software development life cycle, Endres and Rombach attempt to identify the building blocks of a theoretical foundation for software development. In the process, they discuss several interesting—familiar as well as hitherto unfamiliar to many professionals—software engineering rules and laws from which anybody working in software and systems development and maintenance stands to benefit.

The book covers everything from requirements definition through maintenance and retirement. Each chapter presents a set of rules, laws, and hypotheses, discussing their applicability and providing relevant theories and empirical evidence to support their usefulness. For example, Chapter 2—Requirements Definition, Prototyping, and Modeling—presents Glass’s law (“Requirement definitions are the prime sources of project failures”) and Boehm’s law (“Errors are most frequent during the requirements and design activities, and are the most expensive the later they are removed”). It then explains and exemplifies the laws’ applicability, providing evidence in their support and presenting their theoretical basis. The chapter also discusses key activities in this lifecycle phase, such as requirements elicitation and prioritization, knowledge acquisition, feasibility and risk analysis, performance requirements, and requirements traceability.

Subsequent chapters present many other useful laws and rules pertinent to various lifecycle stages such as Constantine’s law (“A structure is stable if cohesion is strong and coupling is...
low”) and Parnas’ law (“Only what is hidden can be changed without risk”) for system design and specification, the Dijkstra-Mills-Wirth law (“Well structured programs have fewer errors and are easier to maintain”) for system construction and composition, Pareto’s law as applicable for software testing, and the three famous Lehman’s laws relating to software maintenance and evolution.

Besides the various lifecycle stages, the book covers project management and business analysis (including the famous Brook’s law); user skills and motivations; technology, architecture, and industry capabilities; and measurements, experimentation, and empirical research. Also, several compact but helpful sections cover preventive versus corrective maintenance, maintenance strategy, cost estimation, and software pricing.

**Problem areas**

One problem area is that Endres and Rombach interchangeably use the terms “law” and “rule” throughout the book, which causes some confusion. Additionally, Chapter 11, Technology, Architecture and Industry Capabilities, isn’t cohesively integrated with the other chapters and thus appears disjointed and force-fitted. Finally, the authors would have done well to highlight the specific contributions of various pioneers when presenting their biographies in Appendix 2.

Notwithstanding these minor problems, *A Handbook of Software and Systems Engineering* is a notable contribution to the evolving profession of software engineering. Furthermore, it represents perhaps the first organized attempt to “discipline” software development by distilling all the relevant rules, laws, and hypotheses. It’s thus a landmark book bound to appeal to those who care about a scientifically grounded approach to software development. Although I imagine the seasoned software development professional already knows some of what’s presented, I’m sure everyone will find the rules and laws discussed useful in their pursuit of software development and management excellence.

**A Practitioner’s Guide to Writing Use Cases**

**Martin Fogarty**


Use cases have become the de facto best practice for capturing requirements in software projects. However, they’re a deceptively simple concept and can easily be misapplied, negating any perceived benefits. Use case proponents often oversell them, resulting in their application in ill-suited areas. Several books are dedicated exclusively to the art and science of writing use cases, all attempting to improve and refine on the work of Ivar Jacobson, the methodology’s founding father. *Use Case Modeling* by Kurt Bittner and Ian Spence is a relatively recent addition. As a member of a software engineering team that has struggled over the past few years with the introduction of a use case-based requirements capture methodology, I was interested to see how well this book treated the subject and whether it dealt with problems we’ve encountered.

**Suits the series**

Having read several books in Addison-Wesley’s Object Technology Series, I was pleased to see that this one followed the same clear format and structure. The book is divided broadly into two parts. The first introduces all the basic concepts of use case modeling, including the whys and wherefores of use cases versus traditional declarative-statement requirements. The second part goes into much more detail on the specifics of writing use cases and the various pitfalls that can occur.

As noted previously, a great danger of use cases is that they seem easy to apply compared to some of Unified Modeling Language’s more arcane notations and concepts. This can lead inexperienced teams to use them without first agreeing on a clearly defined and rigorous methodology. The resulting mess can often turn people off use cases permanently. This is a great shame as use cases, while not a panacea, do represent a significant improvement over traditional requirements capture methods when properly applied. *Use Case Modeling* can help in these situations. The writers obviously have first- or second-hand experience with the major pitfalls and have put together a practitioner’s manual on avoiding them. For example, they devote entire chapters to identifying and documenting use case actors and contexts. Failing to do this correctly completely undermines detailed use case capture because it can cause a team to overlook entire blocks of functionality. I also liked their suggestion of holding use case-modeling workshops as part of the requirements capture process. These are particularly valuable for new teams.

**Take their advice**

If this book had been available when our organization was adopting use case methodology, we would have used large sections of it verbatim to define our new process. It certainly would have helped us avoid early pitfalls. One small quibble I have is that the book doesn’t deal in depth with business use cases and how you can use them to identify system use cases.

I recommend *Use Case Modeling* to any systems engineering team considering this approach. It’s worth noting that this book deals exclusively with use cases. If you’re looking for a broader view of how they fit in the development cycle, try another title in the Object Technology Series, *Use Case Driven Object Modeling With UML* by Doug Rosenberg, as a companion to this book.