A Philosophy Of Software Design

You may be contemplating your first Linux installation. Or you may have been using Linux for years and need to know more about adding a network printer or setting up an FTP server. Running Linux, now in its fifth edition, is the book you'll want on hand in either case. Widely recognized in the Linux community as the ultimate getting-started and problem-solving book, it answers the questions and tackles the configuration issues that frequently plague users, but are seldom addressed in other books. This fifth edition of Running Linux is greatly expanded, reflecting the maturity of the operating system and the teeming wealth of software available for it. Hot consumer topics suchas audio and video playback applications, groupware functionality, and spam filtering are covered, along with the basics in configuration and management that always have made the book popular. Running Linux covers basic communications such as mail, web surfing, and instant messaging, but also delves into the subtleties of network configuration--including dial-up, ADSL, and cable modems--in case you need to set up your network manually. The book canmake you proficient on office suites and personal productivity applications--and also tells you what programming tools are available if you're interested in contributing to these applications. Other new topics in the fifth edition include encrypted email and filesystems, advanced shell techniques, and remote login applications. Classic discussions on booting, package management, kernel recompilation, and X configuration have also been updated. The authors of Running Linux have anticipated problem areas, selected stable and popular solutions, and provided clear instructions to ensure that you'll have a satisfying experience using Linux. The discussion is direct and complete enough to guide novice users, while still providing the additional information experienced users will need to progress in their mastery of Linux. Whether you're using Linux on a home workstation or maintaining a network server, Running Linux will provide expert advice just when you need it.

You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning. The book describes software development, starting from our natural desire to get something of value. Each topic is described with a picture and a few paragraphs. You're invited to think about each topic; to take it in. You'll think about how each step into the process leads to the next. You'll begin to see why Agile methods ask for what they do, and you'll learn why a shallow implementation of Agile can lead to only limited improvement. This is not a detailed map, nor a step-by-step set of instructions for building the perfect project. There is no map or instructions that will do that for you. You need to build your own project, making it a bit more perfect every day. To do that effectively, you need to build up an understanding of the whole process. This book points out the milestones on your journey of understanding the nature of software development done well. It takes you to a location, describes it briefly, and leaves you to explore and fill in your own understanding. What You Need: You'll need your Standard Issue Brain, a bit of curiosity, and a desire to build your own understanding rather than have someone else's detailed ideas poured into your head.

The practical implications of technical debt for the entire software lifecycle; with examples and case studies. Technical debt in software is incurred when developers take shortcuts and make ill-advised technical decisions in the initial phases of a project, only to be confronted with the need for costly and labor-intensive workarounds later. This book offers advice on how to avoid technical debt, how to locate its sources, and how to remove it. It focuses on the practical implications of technical debt for the entire software life cycle, with examples and case studies from companies that range from Boeing to Twitter. Technical debt is normal; it is part of most iterative development processes. But if debt is ignored, over time it may become unmanageably complex, requiring developers to spend all of their effort fixing bugs, with no time to add new features--and after all, new features are what customers really value. The authors explain how to monitor technical debt, how to measure it, and how and when to pay it down. Broadening the conventional definition of technical debt, they cover requirements debt, implementation debt, testing debt, architecture debt, documentation debt, deployment debt, and social debt. They intersperse technical discussions with "Voice of the Practitioner" sidebars that detail real-world experiences with a variety of technical debt issues.

The central topic for this book is the ethics of treating individuals as though they are members of groups. The book raises many interesting questions, including: Why do we feel so much more strongly about discrimination on certain grounds – e.g. of race and sex - than discrimination on other grounds? Are we right to think that discrimination based on these characteristics is especially invidious? What should we think about 'rational discrimination' – 'discrimination' which is based on sound statistics? To take just one of dozens of examples from the book. Suppose a landlord turns away a prospective tenant, because this prospective tenant is of a particular ethnicity – arguing that statistics show that one in four of this group have been shown in the past to default on their rent. That seems clearly unfair to people of this ethnicity. But we are routinely being judged in this way – not just on the basis of our ethnicity, but assumptions are made about us and decisions taken about us based on our gender, religion, job, post-code, hobbies, blood-group, nationality, etc. Now suppose that another landlord turns away a convicted criminal, arguing that one in four of convicted criminals have been shown to be unreliable rent payers. Is our intuition the same as before? Should it be? This book is suitable for all students of philosophy, especially those with an interest in applied ethics.

I owe you a dinner invitation, you owe ten years on your mortgage, and the government owes billions. We speak confidently about these cases of debt, but is that concept clear in its meaning? This book aims to clarify the concept of debt so we can find better answers to important moral and political questions. This book seeks to accomplish two things. The first is to clarify the concept of debt by examining how the word is used in language. The second is to develop a general, principled account of how debts generate genuine obligations. This allows us to avoid settling each case by a bare appeal to moral intuitions, which is what we seem to currently do. It requires a close examination of many institutions, e.g. money, contract law, profit-driven finance, government fiscal operations, and central banking. To properly understand the moral and political nature of debt, we must understand how these institutions have worked, how they do work, and how they might be made to work. There have been many excellent anthropological and sociological studies of debt and its related institutions. Philosophy can contribute to the emerging discussion and help us to keep our language precise and to identify the implicit principles contained in our intuitions.

This book offers an accessible and inclusive overview of the major debates in the philosophy of action. It covers the distinct approaches taken by Donald Davidson, G.E.M. Anscombe, and numerous others to answering questions like "what are intentional actions?" and "how do reasons explain actions?" Further topics include intention, practical knowledge, weakness and strength of will, self-governance, and collective agency. With introductions, conclusions, and annotated suggested reading lists for each of the ten chapters, it is an ideal introduction for advanced undergraduates as well as any philosopher seeking a primer on these issues.

In OBJECT THINKING, esteemed object technologist David West contends that the mindset makes the programmer--not the tools and techniques. Delving into the history, philosophy, and even politics of object-oriented programming, West reveals how the best programmers rely on analysis and conceptualization--on thinking--rather than formal process and methods. Both provocative and pragmatic, this book gives form to what's primarily been an oral tradition among the field's revolutionary thinkers--and it illustrates specific object-behavior practices that you can adopt for true object design and superior results. Gain an in-depth understanding of: Prerequisites and principles of object thinking. Object knowledge implicit in eXtreme Programming (XP) and Agile software development. Object conceptualization and modeling.

Metaphors, vocabulary, and design for object development. Learn viable techniques for: Decomposing complex domains in terms of objects. Identifying object relationships, interactions, and constraints. Relating object behavior to internal structure and implementation design. Incorporating object thinking into XP and Agile practice.

Today, software engineers need to know not only how to program effectively but also how to develop proper engineering practices to make their codebase sustainable and healthy. This book emphasizes this difference between programming and software engineering. How can software engineers manage a living codebase that evolves and responds to changing requirements and demands over the length of its life? Based on their experience at Google, software engineers Titus Winters and Hyrum Wright, along with technical writer Tom Manshreck, present a candid and insightful look at how some of the world's leading practitioners construct and maintain software. This book covers Google's unique engineering culture, processes, and tools and how these aspects contribute to the effectiveness of an engineering organization. You'll explore three fundamental principles that software organizations should keep in mind when designing, architecting, writing, and maintaining code: How time affects the sustainability of software and how to make your code resilient over time How scale affects the viability of software practices within an engineering organization What trade-offs a typical engineer needs to make when evaluating design and development decisions

Presents advice on creating quality design work using repeatable process that solves visual communications issues.

Good software design is simple and easy to understand. Unfortunately, the average computer program today is so complex that no one could possibly comprehend how all the code works. This concise guide helps you understand the fundamentals of good design through scientific laws—principles you can apply to any programming language or project from here to eternity. Whether you're a junior programmer, senior software engineer, or non-technical manager, you'll learn how to create a sound plan for your software project, and make better decisions about the pattern and structure of your system. Discover why good software design has become the missing science Understand the ultimate purpose of software and the goals of good design Determine the value of your design now and in the future Examine real-world examples that demonstrate how a system changes over time Create designs that allow for the most change in the environment with the least change in the software Make easier changes in the future by keeping your code simpler now Gain better knowledge of your software's behavior with more accurate tests

Software Expert Kent Beck Presents a Catalog of Patterns Infinitely Useful for Everyday Programming Great code doesn't just function: it clearly and consistently communicates your intentions, allowing other programmers to understand your code, rely on it, and modify it with confidence. But great code doesn't just happen. It is the outcome of hundreds of small but critical decisions programmers make every single day. Now, legendary software innovator Kent Beck—known worldwide for creating Extreme Programming and pioneering software patterns and test-driven development—focuses on these critical decisions, unearthing powerful "implementation patterns" for writing programs that are simpler, clearer, better organized, and more cost effective. Beck collects 77 patterns for handling everyday programming tasks and writing more readable code. This new collection of patterns addresses many aspects of development, including class, state, behavior, method, collections, frameworks, and more. He uses diagrams, stories, examples, and essays to engage the reader as he illuminates the patterns. You'll find proven solutions for handling everything from naming variables to checking exceptions.

Software legend Max Kanat-Alexander shows you how to succeed as a developer by embracing simplicity, with forty-three essays that will help you really understand the software you work with. About This Book Read and enjoy the superlative writing and insights of the legendary Max Kanat-Alexander Learn and reflect with Max on how to bring simplicity to your software design principles Discover the secrets of rockstar programmers and how to also just suck less as a programmer Who This Book Is For Understanding Software is for every programmer, or anyone who works with programmers. If life is feeling more complex than it should be, and you need to touch base with some clear thinking again, this book is for you. If you need some inspiration and a reminder of how to approach your work as a programmer by embracing some simplicity in your work again, this book is for you. If you're one of Max's followers already, this book is a collection of Max's thoughts selected and curated for you to enjoy and reflect on. If you're new to Max's work, and ready to connect with the power of simplicity again, this book is for you! What You Will Learn See how to bring simplicity and success to your programming world Clues to complexity - and how to build excellent software Simplicity and software design Principles for programmers The secrets of rockstar programmers Max's views and interpretation of the Software industry Why Programmers suck and how to suck less as a programmer Software design in two sentences What is a bug? Go deep into debugging In Detail In Understanding Software, Max Kanat-Alexander, Technical Lead for Code Health at Google, shows you how to bring simplicity back to computer programming. Max explains to you why programmers suck, and how to suck less as a programmer. There's just too much complex stuff in the world. Complex stuff can't be used, and it breaks too easily. Complexity is stupid. Simplicity is smart. Understanding Software covers many areas of programming, from how to write simple code to profound insights into programming, and then how to suck less at what you do! You'll discover the problems with software complexity, the root of its causes, and how to use simplicity to create great software. You'll examine debugging like you've never done before, and how to get a handle on being happy while working in teams. Max brings a selection of carefully crafted essays, thoughts, and advice about working and succeeding in the software industry, from his legendary blog Code Simplicity. Max has crafted forty-three essays which have the power to help you avoid complexity and embrace simplicity, so you can be a happier and more successful developer. Max's technical knowledge, insight, and kindness, has earned him code guru status, and his ideas will inspire you and help refresh your approach to the challenges of being a developer. Style and approach Understanding Software is a new selection of carefully chosen and crafted essays from Max Kanat-Alexander's legendary blog call Code Simplicity. Max's writing and thoughts are great to sit and read cover to cover, or if you prefer you can drop in and see what you discover new every single time!

This book covers all you need to know to model and design software applications from use cases to software architectures in UML and shows how to apply the COMET UML-based modeling and design method to real-world problems. The author describes architectural patterns for various architectures, such as broker, discovery, and transaction patterns for service-oriented architectures, and addresses software quality attributes including maintainability, modifiability, testability, traceability, scalability, reusability, performance, availability, and security. Complete case studies illustrate design issues for different software architectures: a banking system for client/server architecture, an online shopping system for service-oriented architecture, an emergency monitoring system for component-based software architecture, and an automated guided vehicle for real-time software architecture. Organized as an introduction followed by several short, self-contained chapters, the book is perfect for senior undergraduate or graduate courses in software engineering and design, and for experienced software engineers wanting a quick reference at each stage of the analysis, design, and development of large-scale software systems.

A revolutionary concept-based approach to thinking about, designing, and interacting with software As our dependence on technology increases, the design of software matters more than ever before. Why then is so much software flawed? Why hasn't there been a systematic and scalable way to create software that is easy to use, robust, and secure? Examining these issues in depth, The Essence of Software introduces a theory of software design that gives new answers to old questions. Daniel Jackson explains that a software system should be viewed as a collection of interacting concepts, breaking the functionality into manageable parts and providing a new framework for thinking about design. Through this radical and original perspective, Jackson lays out a practical and coherent path, accessible to anyone—from strategist and marketer to UX designer, architect, or programmer—for making software that is empowering, dependable, and a delight to use. Jackson explores every aspect of concepts—what they are and aren't, how to identify them, how to define them, and more—and offers

prescriptive principles and practical tips that can be applied cost-effectively in a wide range of domains. He applies these ideas to contemporary software designs, drawing examples from leading software manufacturers such as Adobe, Apple, Dropbox, Facebook, Google, Microsoft, Twitter, and others. Jackson shows how concepts let designers preserve and reuse design knowledge, rather than starting from scratch in every project. An argument against the status quo and a guide to improvement for both working designers and novices to the field, The Essence of Software brings a fresh approach to software and its creation.

The philosophy of computer science is concerned with issues that arise from reflection upon the nature and practice of the discipline of computer science. This book presents an approach to the subject that is centered upon the notion of computational artefact. It provides an analysis of the things of computer science as technical artefacts. Seeing them in this way enables the application of the analytical tools and concepts from the philosophy of technology to the technical artefacts of computer science. With this conceptual framework the author examines some of the central philosophical concerns of computer science including the foundations of semantics, the logical role of specification, the nature of correctness, computational ontology and abstraction, formal methods, computational epistemology and explanation, the methodology of computer science, and the nature of computation. The book will be of value to philosophers and computer scientists.

Play is a vital component of the social life and well-being of both children and adults. This book examines the concept of play and considers a variety of the related philosophical issues. It also includes meta-analyses from a range of philosophers and theorists, as well as an exploration of some key applied ethical considerations. The main objective of The Philosophy of Play is to provide a richer understanding of the concept and nature of play and its relation to human life and values, and to build disciplinary and paradigmatic bridges between scholars of philosophy and scholars of play. Including specific chapters dedicated to children and play, and exploring the work of key thinkers such as Plato, Sartre, Wittgenstein, Gadamer, Deleuze and Nietzsche, this book is invaluable reading for any advanced student, researcher or practitioner with an interest in education, playwork, leisure studies, applied ethics or the philosophy of sport.

"This book offers a high interdisciplinary exchange of ideas pertaining to the philosophy of computer science, from philosophical and mathematical logic to epistemology, engineering, ethics or neuroscience experts and outlines new problems that arise with new tools"--Provided by publisher.

Although scholarship in philosophy of action has grown in recent years, there has been little work explicitly dealing with the role of time in agency, a role with great significance for the study of action. As the articles in this collection demonstrate, virtually every fundamental issue in the philosophy of action involves considerations of time. The four sections of this volume address the metaphysics of action, diachronic practical rationality, the relation between deliberation and action, and the phenomenology of agency, providing an overview of the central developments in each area with an emphasis on the role of temporality. Including contributions by established, rising, and new voices in the field, Time and the Philosophy of Action brings analytic work in philosophy of action together with contributions from continental philosophy and cognitive science to elaborate the central thesis that agency not only develops in time but is shaped by it at every level.

Awareness of design smells – indicators of common design problems – helps developers or software engineers understand mistakes made while designing, what design principles were overlooked or misapplied, and what principles need to be applied properly to address those smells through refactoring. Developers and software engineers may "know" principles and patterns, but are not aware of the "smells" that exist in their design because of wrong or mis-application of principles or patterns. These smells tend to contribute heavily to technical debt – further time owed to fix projects thought to be complete – and need to be addressed via proper refactoring. Refactoring for Software Design Smells presents 25 structural design smells, their role in identifying design issues, and potential refactoring solutions. Organized across common areas of software design, each smell is presented with diagrams and examples illustrating the poor design practices and the problems that result, creating a catalog of nuggets of readily usable information that developers or engineers can apply in their projects. The authors distill their research and experience as consultants and trainers, providing insights that have been used to improve refactoring and reduce the time and costs of managing software projects. Along the way they recount anecdotes from actual projects on which the relevant smell helped address a design issue. Contains a comprehensive catalog of 25 structural design smells (organized around four fundamental design principles) that contribute to technical debt in software projects Presents a unique naming scheme for smells that helps understand the cause of a smell as well as points toward its potential refactoring Includes illustrative examples that showcase the poor design practices underlying a smell and the problems that result Covers pragmatic techniques for refactoring design smells to manage technical debt and to create and maintain high-quality software in practice Presents insightful anecdotes and case studies drawn from the trenches of real-world projects

Summary Type-Driven Development with Idris, written by the creator of Idris, teaches you how to improve the performance and accuracy of your programs by taking advantage of a state-of-the-art type system. This book teaches you with Idris, a language designed to support type-driven development. Purchase of the print book includes a free eBook in PDF, Kindle, and ePub formats from Manning Publications. About the Technology Stop fighting type errors! Type-driven development is an approach to coding that embraces types as the foundation of your code - essentially as built-in documentation your compiler can use to check data relationships and other assumptions. With this approach, you can define specifications early in development and write code that's easy to maintain, test, and extend. Idris is a Haskell-like language with first-class, dependent types that's perfect for learning type-driven programming techniques you can apply in any codebase. About the Book Type-Driven Development with Idris teaches you how to improve the performance and accuracy of your code by taking advantage of a state-of-the-art type system. In this book, you'll learn type-driven development of real-world software, as well as how to handle side effects, interaction, state, and concurrency. By the end, you'll be able to develop robust and verified software in Idris and apply type-driven development methods to other languages. What's Inside Understanding dependent types Types as first-class language constructs Types as a guide to program construction Expressing relationships between data About the Reader Written for programmers with knowledge of functional programming concepts. About the Author Edwin Brady leads the design and implementation of the Idris language. Table of Contents PART 1 - INTRODUCTION Overview Getting started with

IdrisPART 2 - CORE IDRIS Interactive development with types User-defined data types Interactive programs: input and output processing Programming with first-class types Interfaces: using constrained generic types Equality: expressing relationships between data Predicates: expressing assumptions and contracts in types Views: extending pattern matching PART 3 - IDRIS AND THE REAL WORLD Streams and processes: working with infinite data Writing programs with state State machines: verifying protocols in types Dependent state machines: handling feedback and errors Type-safe concurrent programming

This volume provides the reader with an integrated overview of state-of-the-art research in philosophy and ethics of design in engineering and architecture. It contains twenty-five essays that focus on engineering designing in its traditional sense, on designing in novel engineering domains, and on architectural and environmental designing. This volume enables the reader to overcome the traditional separation between engineering designing and architectural designing.

A Philosophy of Landscape Construction outlines a philosophy of values in landscape construction, demonstrating how integral structures, such as pavements and walls, constitute a key element to how people interact with and inhabit the final design. The book discusses how these structures enable, assist and care for people, negotiating between the dynamic processes of site ecosystems and the soil on which they are founded. They articulate spatial, functional, cultural and ecological meanings. Within this theoretical framework, designers will learn to recognize and insert a set of core values into the most technical design stages to reach their full potential. By offering a new perspective on landscape construction, moving away from the exclusively technical characteristics, this book allows landscape architects to realise the ideal vision for their designs. It is abundantly illustrated with examples from which designers can learn both successes and failures and will be an essential companion to any study of built landscapes.

This book is a critical introduction to code and software that develops an understanding of its social and philosophical implications in the digital age. Written specifically for people interested in the subject from a non-technical background, the book provides a lively and interesting analysis of these new media forms.

In the debate leading up to the EU referendum in the United Kingdom, the British politician Michael Gove declared that "people in this country have had enough of experts". In the 2016 Presidential campaign in the United States, Donald Trump waged a war against the very idea of expertise. Yet if you are worried about your child's behaviour, don't know which laptop to buy, or just want to get fit, the answer is easy: ask an expert. Where do we draw the line? Why do we appear to know more and more collectively, yet less and less individually? Has expertise painted itself into a corner? Can we defend both science and common sense? In this engaging and much-needed book Jan Bransen explores these important questions and more. He argues that the rise of behavioural sciences has caused a sea change in the relationship between science and common sense. He shows how - as recently as the 1960s - common sense and science were allies in the battle against ignorance, but that since then populism and chauvinism have claimed common sense as their own. Bransen argues that common sense is a collection of interrelated skills that draw on both an automatic pilot and an investigative attitude where we ask ourselves the right questions. It is the very attitude of open-minded inquiry and questioning that Bransen believes we are at risk of losing in the face of an army of experts. Drawing on fascinating examples such as language and communication, money, the imaginary world of Endoxa, domestic violence, and quality of life, Don't be Fooled: A Philosophy of Common Sense is a brilliant and wry defence of a skill that is a vital part of being human.

Design and Spirituality examines the philosophical context of our current situation and its implications for design. It explores how modernity and our constricted notions of progress have contributed to today's crisis of values, and argues for a re-establishment and re-affirmation of self-transcending priorities, together with an ethos of moderation and sufficiency. A wide range of topics are covered, including material culture and spiritual teachings; sustainability and the spiritual perspective; traditional and indigenous knowledge; technology and spirituality; notions of meaningful design; and how particular material things can have deeper, symbolic significance. There are also reflections on areas such as the language of design; busyness and its relationship to wisdom; design and social disparity; and traditional sacred practices. While not avoiding issues that are controversial, and sometimes hard-hitting, Design and Spirituality gets to the heart of the key issues affecting us today and presents them in a highly readable and accessible format. The author is a leading thinker in the field and he presents his arguments in a manner that invites the reader to reflect and think about where we are going, why we are going there and what really matters.

Are you working on a codebase where cost overruns, death marches, and heroic fights with legacy code monsters are the norm? Battle these adversaries with novel ways to identify and prioritize technical debt, based on behavioral data from how developers work with code. And that's just for starters. Because good code involves social design, as well as technical design, you can find surprising dependencies between people and code to resolve coordination bottlenecks among teams. Best of all, the techniques build on behavioral data that you already have: your version-control system. Join the fight for better code! Use statistics and data science to uncover both problematic code and the behavioral patterns of the developers who build your software. This combination gives you insights you can't get from the code alone. Use these insights to prioritize refactoring needs, measure their effect, find implicit dependencies between different modules, and automatically create knowledge maps of your system based on actual code contributions. In a radical, much-needed change from common practice, guide organizational decisions with objective data by measuring how well your development teams align with the software architecture. Discover a comprehensive set of practical analysis techniques based on version-control data, where each point is illustrated with a case study from a real-world codebase. Because the techniques are language neutral, you can apply them to your own code no matter what programming language you use. Guide organizational decisions with objective data by measuring how well your development teams align with the software architecture. Apply research findings from social psychology to software development, ensuring you get the tools you need to coach your organization towards better code. If you're an experienced programmer, software architect, or technical manager, you'll get a new perspective that will change how you work with code. What You Need: You don't have to install anything to follow along in the book. TThe case studies in the book use well-known open source projects hosted on GitHub. You'll use CodeScene, a free software analysis tool for open source projects, for the case studies. We also discuss alternative tooling options where they exist.

Strategies for building large systems that can be easily adapted for new situations with only minor programming modifications. Time pressures encourage programmers to write code that works well for a narrow purpose, with no room to grow. But the best systems are evolvable; they can be adapted for new situations by adding code, rather than changing the existing code. The authors describe techniques they have found effective--over their combined 100-plus years of programming experience--that will help programmers avoid programming themselves into corners. The authors explore ways to enhance flexibility by: • Organizing systems using combinators to compose mix-and-match parts, ranging from small functions to whole arithmetics, with standardized interfaces • Augmenting data with independent annotation layers, such as units of measurement or provenance • Combining independent pieces of partial information using unification or propagation • Separating control structure from problem domain with domain models, rule systems and pattern matching, propagation, and dependency-directed backtracking • Extending the programming language, using dynamically extensible evaluators

With this practical book, architects, CTOs, and CIOs will learn a set of patterns for the practice of architecture, including analysis, documentation, and communication. Author Eben Hewitt shows you how to create holistic and thoughtful technology plans, communicate them clearly, lead people toward the vision, and become a great architect or Chief Architect. This book covers each key aspect of architecture comprehensively, including how to incorporate business architecture, information architecture, data architecture, application (software) architecture together to have the best chance for the system's success. Get a practical set of proven architecture practices focused on shipping great products using architecture Learn how architecture works effectively with development teams, management, and product management teams through the value chain Find updated special coverage on machine learning architecture Get usable templates to start incorporating into your teams immediately Incorporate business architecture, information architecture, data architecture, and application (software) architecture together

Right Your Software and Transform Your Career Righting Software presents the proven, structured, and highly engineered approach to software design that renowned architect Juval Löwy has practiced and taught around the world. Although companies of every kind have successfully implemented his original design ideas across hundreds of systems, these insights have never before appeared in print. Based on first principles in software engineering and a comprehensive set of matching tools and techniques, Löwy's methodology integrates system design and project design. First, he describes the primary area where many software architects fail and shows how to decompose a system into smaller building blocks or services, based on volatility. Next, he shows how to flow an effective project design from the system design; how to accurately calculate the project duration, cost, and risk; and how to devise multiple execution options. The method and principles in Righting Software apply regardless of your project and company size, technology, platform, or industry. Löwy starts the reader on a journey that addresses the critical challenges of software development today by righting software systems and projects as well as careers—and possibly the software industry as a whole. Software professionals, architects, project leads, or managers at any stage of their career will benefit greatly from this book, which provides guidance and knowledge that would otherwise take decades and many projects to acquire. Register your book for convenient access to downloads, updates, and/or corrections as they become available. See inside book for details.

Childhood looms large in our understanding of human life, as a phase through which all adults have passed. Childhood is foundational to the development of selfhood, the formation of interests, values and skills and to the lifespan as a whole. Understanding what it is like to be a child, and what differences childhood makes, are thus essential for any broader understanding of the human condition. The Routledge Handbook of the Philosophy of Childhood and Children is an outstanding reference source for the key topics, problems and debates in this crucial and exciting field and is the first collection of its kind. Comprising over thirty chapters by a team of international contributors the Handbook is divided into five parts: · Being a child · Childhood and moral status · Parents and children · Children in society · Children and the state. Questions covered include: What is a child? Is childhood a uniquely valuable state, and if so why? Can we generalize about the goods of childhood? What rights do children have, and are they different from adults' rights? What (if anything) gives people a right to parent? What role, if any, ought biology to play in determining who has the right to parent a particular child? What kind of rights can parents legitimately exercise over their children? What roles do relationships with siblings and friends play in the shaping of childhoods? How should we think about sexuality and disability in childhood, and about racialised children? How should society manage the education of children? How are children's lives affected by being taken into social care? The Routledge Handbook of the Philosophy of Childhood and Children is essential reading for students and researchers in philosophy of childhood, political philosophy and ethics as well as those in related disciplines such as education, psychology, sociology, social policy, law, social work, youth work, neuroscience and anthropology.

The rise of classic Euro-American philosophy of technology in the 1950s originally emphasized the importance of technologies as material entities and their mediating influence within human experience. Recent decades, however, have witnessed a subtle shift toward reflection on the activity from which these distinctly modern artifacts emerge and through which they are engaged and managed, that is, on engineering. What is engineering? What is the meaning of engineering? How is engineering related to other aspects of human existence? Such basic questions readily engage all major branches of philosophy --- ontology, epistemology, ethics, political philosophy, and aesthetics --- although not always to the same degree. The historico-philosophical and critical reflections collected here record a series of halting steps to think through engineering and the engineered way of life that we all increasingly live in what has been called the Anthropocene. The aim is not to promote an ideology for engineering but to stimulate deeper reflection among engineers and non-engineers alike about some basic challenges of our engineered and engineering lifeworld.

With a spice of wit and illuminating illustration, this collection of 75 short pieces deals with topics in the field of software requirements analysis, specifications and design. The author emphasizes the need to structure and analyze problems, not just specify a solution.

An engaging, illustrated collection of insights revealing the practices and principles that expert software designers use to create great software. What makes an expert software designer? It is more than experience or innate ability. Expert software designers have specific habits, learned practices, and observed principles that they apply deliberately during their design work. This book offers sixty-six insights, distilled from years of studying experts at work, that capture what successful software designers actually do to create great software. The book presents these insights in a series of two-page illustrated spreads, with the principle and a short explanatory text on one page, and a drawing on the facing page. For example, "Experts generate alternatives" is illustrated

by the same few balloons turned into a set of very different balloon animals. The text is engaging and accessible; the drawings are thought-provoking and often playful. Organized into such categories as "Experts reflect," "Experts are not afraid," and "Experts break the rules," the insights range from "Experts prefer simple solutions" to "Experts see error as opportunity." Readers learn that "Experts involve the user"; "Experts take inspiration from wherever they can"; "Experts design throughout the creation of software"; and "Experts draw the problem as much as they draw the solution." One habit for an aspiring expert software designer to develop would be to read and reread this entertaining but essential little book. The insights described offer a guide for the novice or a reference for the veteran—in software design or any design profession. A companion web site provides an annotated bibliography that compiles key underpinning literature, the opportunity to suggest additional insights, and more.

The Philosophy of Design is an introduction to the fundamental philosophical issues raised by the contemporary practice of design. The first book to systematically examine design from the perspective of contemporary philosophy, it offers a broad perspective, ranging across key philosophical areas such as aesthetics, epistemology, metaphysics and ethics. The first part of the book explores central issues about the nature of design and its products, and the rationality of design methods. A central theme is that Modernist ideas, such as those offered by Loos and Gropius, provide important responses to these philosophical issues. In the second part of the book, these Modernist ideas serve as touchstones in the exploration of key issues for design, including: the place of aesthetics in design; design's relation to personal expression; the meaning of function; and design's relation to consumerism. The social responsibility of designers, and the impact of design practice on ethical reasoning are also discussed. Written in an accessible style, The Philosophy of Design presents a new perspective on design and a provocative reassessment of the Modernist legacy. It will engage students and designers with current philosophical debates, helping them to bring into clearer focus the meaning of contemporary design, and its unique challenges and possibilities.

Today, "simple living" is a rallying cry for anti-consumerists, environmentalists, and anyone concerned with humanity's effect on the planet. But what is so revolutionary about a simple life? And why are we so fascinated with simplicity today? A Philosophy of Simple Living charts the ideas, motivations, and practices of simplicity from antiquity to the present day. Bringing together an array of people, practices, and movements, from Henry David Thoreau to Steve Jobs, and from Cynics and Shakers to the "slow movement," voluntary simplicity, and degrowth, this book is as comprehensive as it is concise. Written in elegant, spare prose, A Philosophy of Simple Living will be of great benefit to all who wish to declutter and pare back their complicated, modern lives.

Peter Seibel interviews 15 of the most interesting computer programmers alive today in Coders at Work, offering a companion volume to Apress's highly acclaimed best-seller Founders at Work by Jessica Livingston. As the words "at work" suggest, Peter Seibel focuses on how his interviewees tackle the day-to-day work of programming, while revealing much more, like how they became great programmers, how they recognize programming talent in others, and what kinds of problems they find most interesting. Hundreds of people have suggested names of programmers to interview on the Coders at Work web site: www.codersatwork.com. The complete list was 284 names. Having digested everyone's feedback, we selected 15 folks who've been kind enough to agree to be interviewed: Frances Allen: Pioneer in optimizing compilers, first woman to win the Turing Award (2006) and first female IBM fellow Joe Armstrong: Inventor of Erlang Joshua Bloch: Author of the Java collections framework, now at Google Bernie Cosell: One of the main software guys behind the original ARPANET IMPs and a master debugger Douglas Crockford: JSON founder, JavaScript architect at Yahoo! L. Peter Deutsch: Author of Ghostscript, implementer of Smalltalk-80 at Xerox PARC and Lisp 1.5 on PDP-1 Brendan Eich: Inventor of JavaScript, CTO of the Mozilla Corporation Brad Fitzpatrick: Writer of LiveJournal, OpenID, memcached, and Perlbal Dan Ingalls: Smalltalk implementor and designer Simon Peyton Jones: Coinventor of Haskell and lead designer of Glasgow Haskell Compiler Donald Knuth: Author of The Art of Computer Programming and creator of TeX Peter Norvig: Director of Research at Google and author of the standard text on AI Guy Steele: Coinventor of Scheme and part of the Common Lisp Gang of Five, currently working on Fortress Ken Thompson: Inventor of UNIX Jamie Zawinski: Author of XEmacs and early Netscape/Mozilla hacker

Summary Grokking Deep Learning teaches you to build deep learning neural networks from scratch! In his engaging style, seasoned deep learning expert Andrew Trask shows you the science under the hood, so you grok for yourself every detail of training neural networks. Purchase of the print book includes a free eBook in PDF, Kindle, and ePub formats from Manning Publications. About the Technology Deep learning, a branch of artificial intelligence, teaches computers to learn by using neural networks, technology inspired by the human brain. Online text translation, self-driving cars, personalized product recommendations, and virtual voice assistants are just a few of the exciting modern advancements possible thanks to deep learning. About the Book Grokking Deep Learning teaches you to build deep learning neural networks from scratch! In his engaging style, seasoned deep learning expert Andrew Trask shows you the science under the hood, so you grok for yourself every detail of training neural networks. Using only Python and its math-supporting library, NumPy, you'll train your own neural networks to see and understand images, translate text into different languages, and even write like Shakespeare! When you're done, you'll be fully prepared to move on to mastering deep learning frameworks. What's inside The science behind deep learning Building and training your own neural networks Privacy concepts, including federated learning Tips for continuing your pursuit of deep learning About the Reader For readers with high school-level math and intermediate programming skills. About the Author Andrew Trask is a PhD student at Oxford University and a research scientist at DeepMind. Previously, Andrew was a researcher and analytics product manager at Digital Reasoning, where he trained the world's largest artificial neural network and helped guide the analytics roadmap for the Synthesys cognitive computing platform. Table of Contents Introducing deep learning: why you should learn it Fundamental concepts: how do machines learn? Introduction to neural prediction: forward propagation Introduction to neural learning: gradient descent Learning multiple weights at a time: generalizing gradient descent Building your first deep neural network: introduction to backpropagation How to picture neural networks: in your head and on paper Learning signal and ignoring noise:introduction to regularization and batching Modeling probabilities and nonlinearities: activation functions Neural learning about edges and corners: intro to convolutional neural networks Neural networks that understand language: king - man + woman == ? Neural networks that write like Shakespeare: recurrent layers for variable-length data Introducing automatic optimization: let's build a deep learning framework Learning to write like Shakespeare: long short-term memory Deep learning on unseen data: introducing federated learning Where to go from here: a brief guide

Many claims are made about how certain tools, technologies, and practices improve software development. But which claims are verifiable, and which are merely wishful thinking? In this book, leading thinkers such as Steve McConnell, Barry Boehm, and Barbara Kitchenham offer essays that uncover the truth and unmask myths commonly held among the software development

community. Their insights may surprise you. Are some programmers really ten times more productive than others? Does writing tests first help you develop better code faster? Can code metrics predict the number of bugs in a piece of software? Do design patterns actually make better software? What effect does personality have on pair programming? What matters more: how far apart people are geographically, or how far apart they are in the org chart? Contributors include: Jorge Aranda Tom Ball Victor R. Basili Andrew Begel Christian Bird Barry Boehm Marcelo Cataldo Steven Clarke Jason Cohen Robert DeLine Madeline Diep Hakan Erdogmus Michael Godfrey Mark Guzdial Jo E. Hannay Ahmed E. Hassan Israel Herraiz Kim Sebastian Herzig Cory Kapser Barbara Kitchenham Andrew Ko Lucas Layman Steve McConnell Tim Menzies Gail Murphy Nachi Nagappan Thomas J. Ostrand Dewayne Perry Marian Petre Lutz Prechelt Rahul Premraj Forrest Shull Beth Simon Diomidis Spinellis Neil Thomas Walter Tichy Burak Turhan Elaine J. Weyuker Michele A. Whitecraft Laurie Williams Wendy M. Williams Andreas Zeller Thomas Zimmermann

Despite the proliferation of video games in the twenty-first century, the theory of game design is largely underdeveloped, leaving designers on their own to understand what games really are. Helping you produce better games, Game Design Theory: A New Philosophy for Understanding Games presents a bold new path for analyzing and designing games. The author offers a radical yet reasoned way of thinking about games and provides a holistic solution to understanding the difference between games and other types of interactive systems. He clearly details the definitions, concepts, and methods that form the fundamentals of this philosophy. He also uses the philosophy to analyze the history of games and modern trends as well as to design games. Providing a robust, useful philosophy for game design, this book gives you real answers about what games are and how they work. Through this paradigm, you will be better equipped to create fun games.

Copyright: b3beba342897bc7b986a791b790b0019