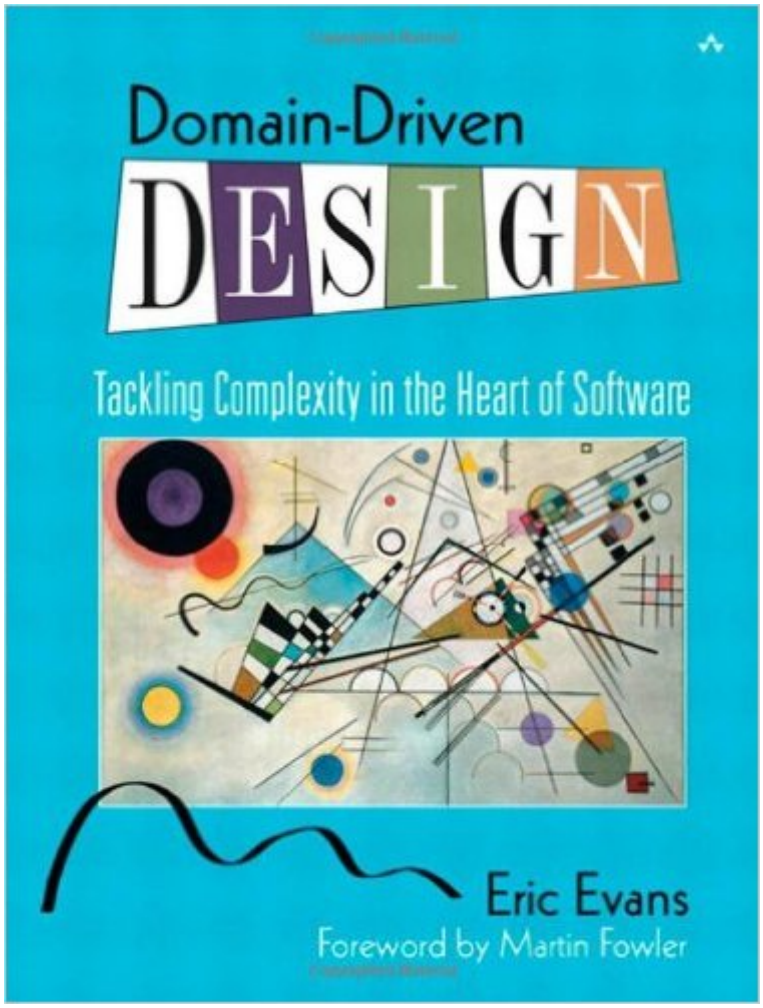
Domain-Driven Design: Tackling Complexity In The Heart Of Software



Synopsis

Title: Domain-Driven Design(Tackling Complexity in the Heart of Software) Binding: Hardcover

Author: EricEvans Publisher: Addison-WesleyProfessional

Book Information

Hardcover: 560 pages

Publisher: Addison-Wesley Professional; 1 edition (August 30, 2003)

Language: English

ISBN-10: 0321125215

ISBN-13: 978-0321125217

Product Dimensions: 7.4 x 1.5 x 9.3 inches

Shipping Weight: 2.8 pounds (View shipping rates and policies)

Average Customer Review: 4.3 out of 5 starsÂ Â See all reviewsÂ (97 customer reviews)

Best Sellers Rank: #27,788 in Books (See Top 100 in Books) #11 inÂ Books > Computers & Technology > Computer Science > Systems Analysis & Design #12 inÂ Books > Textbooks > Computer Science > Object-Oriented Software Design #26 inÂ Books > Textbooks > Computer Science > Software Design & Engineering

Customer Reviews

If you have even been involved in a software project (a) as a developer and did not know what the end product is going to be used for or how it will be used or (b) as an architect who spent countless hours with your stakeholders and domain experts trying to figure out how to go about architecting your application, then you should read this book. Read it again after you have read it for the first time. This book is packed with pointers, information, tips, how-tos, "down to earth" practical samples, and even conversational examples that one could have while gathering requirements. Evans in his book fills a wide gap that we all tend to come across while designing software applications.There are many software engineering processes out there, and each one tries to tackle the complexities of designing software applications for a given domain in its own way. Evans recognizes the tools and the processes that are popular in the industry, UML, Agile, and focuses on some aspects of the software engineering process that we tend to miss. He starts the book by talking about the importance of creating and having a Ubiquitous Language. There is a similar concept in the RUP, but not emphasizes as much - or at all. Evans goes into a great detail on why, from the inception of a project, it is important to have a common language and gives many pointers on what makes up the Ubiquitous Language for each project:"Use the model as the backbone of a

language. Commit the team to exercising that language relentlessly within the team and the in the code. Use the same language in diagrams, writing, and especially speech."Parts II-IV of the book put domain-driven design in perspective, and show the reader thru examples and patterns, architectural patterns, design patterns and process patterns, the importance of having a consistent model that maps to the domain and how to go about achieving such model. In an essence, "Model-Driven Design discards the dichotomy of analysis model and design to search out a single model that serves both purposes".Part II of the book, introduces the building blocks of a Model-Driven Design. This section, as with the others, takes popular patterns from the Gamma, Flower, or others and applies them to the topic at hand - Model-Driven Design. In that aspect, the reader can easily follow the text and relate to topic at hand. Evans uses the ever-popular Model-View-Controller (MVC) design pattern to get things going in part II. He then goes off to explain why the layered architecture approach is an important aspect of a Domain-Driven Design and how it would makes things simpler:"[Layered Architecture] allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work."The author then goes into great detail in explaining the elements that express a model:1) Entities: An object that is tracked thru different states or even across different implementations.2) Value Objects: An attribute that describes the state of something else.3) Services: Aspects of domain that are expressed as actions or operations, rather than objects.4) Packages: Organize the objects and services.What do you want to do after you have designed such elements? The creation and life cycle management of objects are discussed next in this book. Three patterns, mostly from the Gamma book, are used to manage the life cycle of objects:1) Aggregates.2) Factories.3) Repositories.Aggregates represent the hierarchy of objects or services and their interactions. Factories and Repositories operate of Aggregates and encapsulate the complexity of specific life cycle transitions.Part III of the book talks about the things developers and architects need to do to achieve a Supple Design. Refactoring over and over represents the topic in this section: "Each refinement of code and model gives developers a clearer view"The author talks about a breakthrough point during the design that the "designers see the light" and both the domain experts and the designers, after many iterations, have finally come to this higher level of understanding of the domain and the value of refactoring exponentially increases after that.Part IV of this book talks about a very important topic that we all have struggled with one time or another: the ability of the model and the modeling process to scale up to very complicated domains. It is great that we can model a small domain, but one goes about modeling an enterprise, which is most likely, too complex to model as a single unit? Low-coupling and high cohesion still applies here, but the goal is to not

loose anything during the integration process. The author goes in to a great detail in this part to emphasize that even in large circumstances such as modeling an enterprise, every decision must have a direct impact on system development. Three different themes are covered in this section in order to assist with modeling of large units:1) Context: the model has to be logically consistent throughout, without contradictory or overlapping definitions. For this theme, the author introduces the concept of a Bonded Context- a way that relationship to other context are defined a overlapping is then avoided.2) Distillation: Reducing the clutter and focusing attention appropriately.3) Large-scale Structure. The concept of Responsibility layers are introducedIn summary, Evans did a great job in writing this book, and filling it with useful ways of designing and architecting software applications that target a domain, which in most cases we do not know much about.

I bought this book due in part to the glowing reviews here on so I feel a duty to inject a bit of skepticism, now that I've read it.5 stars for a technical book indicates to me a book of profound quality that really breaks through with penetrating insights -- The kind of book that makes me think, "Wow, this book has really brought my development practice into a renewed, sharper focus." It doesn't necessarily have to provide radically new material, but it does have to package whatever material it contains in a way that causes the gears in my head to shift around and reorganize themselves. Design Patterns is such a book. XP Explained is such a book. I don't think this one qualifies.Some good points: The author makes a good case for agile development/extreme programming (close relationship with the customer, unit tests, refactoring...). He seems to believe there may be a tendency to over-emphasize the importance of code and to neglect design in such practices, which may or may not be true in industry at large. But in any case, his major thesis is that it is also important to consider the overall domain model and how well-aligned it is to the goals of the business. He proposes developing a common ("ubiqitous") language between developers and business users, and to unify the various traditional views of a software system (requirements, analysis model, design model, etc..) into one. The advice is quite wholesome and will hopefully promote bringing some harmony between the agile camp and the adherents of high-ceremony approaches such as RUP and CMM.Some bad points: The book is rather wordy, and a lot of common-sense ideas are repeated at length. I don't feel that the patterns in the book are much more than re-statements of basic principles of OO design. I am not convinced that giving every possible variation on OO programming a fancy name is particularly helpful. Most of the patterns in this book come down to "produce a clean design that removes duplication and attempts to match the business domain." If you're new to OO, I suggest you'd be much better off reading some other

books, such as GoF's Design Patterns, Fowler's Refactoring, Page-Jones' Object-Oriented Design in UML, and Kent Beck's XP Explained.I give this book 3 stars because it's not a bad thing to read a book that makes you think about the importance of the business domain when programming. It's true that this emphasis, while fairly basic, does get lost in a world where specific technologies dominate good design and common sense. I don't think this book can really hurt -- although I have found the "declarative" approach it mentions can be very dangerous in inexperienced hands and can produce utterly unmaintainable code. It's not a bad effort, but it's not an earth shattering revelation either.

I think that this book along with Robert Martin's "Agile Software Development, Principles, Patterns, and Practices" and Martin Fowler's "Refactoring" are perhaps the three most fundamental prerequisites for making the leap in knowledge and maturity from object-oriented programming to true proficiency in object-oriented design. The books from Martin and Fowler cover the software solution design space and the core principles and patterns for making code that is resilient to change and easy to maintain. Eric Evans book covers the problem domain space and the abstraction skills that free programmers to "break out of the box" of the implementation domain and solution objects into the critical area the business domain and corresponding domain objects.I once led a young software team and tried to convey the need for and essence of these skills to them, but I didnt have the right words and terms to do it for their level of experience. I wish this book had been available to me then because I think it would have made a real difference for that team.

Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems Lessons Learned in Software Testing: A Context-Driven Approach Just Enough Software Architecture: A Risk-Driven Approach Software Process Design: Out of the Tar Pit (Mcgraw-Hill International Software Quality Assurance) Constraint-Based Design Recovery for Software Reengineering: Theory and Experiments (International Series in Software Engineering) Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures The Domain Name Handbook; High Stakes and Strategies in Cyberspace DNSSEC Mastery: Securing the Domain Name System with BIND Domain Name Profits