From the Library of Javier Acero

**Praise for *Implementing Domain-Driven Design***

“With *Implementing Domain-Driven Design*, Vaughn has made an important con tribution not only to the literature of the Domain-Driven Design community, but also to the literature of the broader enterprise application architecture field. In key chap ters on Architecture and Repositories, for example, Vaughn shows how DDD fits with the expanding array of architecture styles and persistence technologies for enterprise applications—including SOA and REST, NoSQL and data grids—that has emerged in the decade since Eric Evans’ seminal book was first published. And, fittingly, Vaughn illuminates the blocking and tackling of DDD—the implementation of entities, value objects, aggregates, services, events, factories, and repositories—with plentiful exam ples and valuable insights drawn from decades of practical experience. In a word, I would describe this book as *thorough*. For software developers of all experience levels looking to improve their results, and design and implement domain-driven enterprise applications consistently with the best current state of professional practice, *Imple menting Domain-Driven Design* will impart a treasure trove of knowledge hard won within the DDD and enterprise application architecture communities over the last cou ple decades.”

—Randy Stafford, Architect At-Large, Oracle Coherence Product Development

“Domain-Driven Design is a powerful set of thinking tools that can have a profound impact on how effective a team can be at building software-intensive systems. The thing is that many developers got lost at times when applying these thinking tools and really needed more concrete guidance. In this book, Vaughn provides the missing links between theory and practice. In addition to shedding light on many of the misunder

stood elements of DDD, Vaughn also connects new concepts like Command/Query Responsibility Segregation and Event Sourcing that many advanced DDD practitioners have used with great success. This book is a must-read for anybody looking to put DDD into practice.”

—Udi Dahan, Founder of NServiceBus

“For years, developers struggling to practice Domain-Driven Design have been wishing for more practical help in actually implementing DDD. Vaughn did an excellent job in closing the gap between theory and practice with a complete implementation reference. He paints a vivid picture of what it is like to do DDD in a contemporary project, and provides plenty of practical advice on how to approach and solve typical challenges occurring in a project life cycle.”

—Alberto Brandolini, DDD Instructor, Certified by Eric Evans and

Domain Language, Inc.

“*Implementing Domain-Driven Design* does a remarkable thing: it takes a sophisti cated and substantial topic area in DDD and presents it clearly, with nuance, fun and finesse. This book is written in an engaging and friendly style, like a trusted advisor giving you expert counsel on how to accomplish what is most important. By the time you finish the book you will be able to begin applying all the important concepts of

From the Library of Javier Acero

DDD, and then some. As I read, I found myself highlighting many sections . . . I will be referring back to it, and recommending it, often.”

—Paul Rayner, Principal Consultant & Owner, Virtual Genius, LLC., DDD Instruc tor, Certified by Eric Evans and Domain Language, Inc., DDD Denver Founder and Co-leader

“One important part of the DDD classes I teach is discussing how to put all the ideas and pieces together into a full blown working implementation. With this book, the DDD community now has a comprehensive reference that addresses this in detail. *Implementing Domain-Driven Design* deals with all aspects of building a system using DDD, from getting the small details right to keeping track of the big picture. This is a great reference and an excellent companion to Eric Evans seminal DDD book.”

—Patrik Fredriksson, DDD Instructor, Certified by Eric Evans and

Domain Language, Inc.

“If you care about software craftsmanship—and you should—then Domain-Driven Design is a crucial skill set to master and *Implementing Domain-Driven Design* is the fast path to success. *IDDD* offers a highly readable yet rigorous discussion of DDD’s strategic and tactical patterns that enables developers to move immediately from under

standing to action. Tomorrow’s business software will benefit from the clear guidance provided by this book.”

—Dave Muirhead, Principal Consultant, Blue River Systems Group

“There’s theory and practice around DDD that every developer needs to know, and this is the missing piece of the puzzle that puts it all together. Highly recommended!” —Rickard Öberg, Java Champion and Developer at Neo Technology

“In *IDDD*, Vaughn takes a top-down approach to DDD, bringing strategic patterns such as bounded context and context maps to the fore, with the building block patterns of entities, values and services tackled later. His book uses a case study throughout, and to get the most out of it you’ll need to spend time grokking that case study. But if you do you’ll be able to see the value of applying DDD to a complex domain; the fre

quent sidenotes, diagrams, tables, and code all help illustrate the main points. So if you want to build a solid DDD system employing the architectural styles most commonly in use today, Vaughn’s book comes recommended.”

—Dan Haywood, author of *Domain-Driven Design with Naked Objects*

“This book employs a top-down approach to understanding DDD in a way that fluently connects strategic patterns to lower level tactical constraints. Theory is coupled with guided approaches to implementation within modern architectural styles. Throughout the book, Vaughn highlights the importance and value of focusing on the business domain all while balancing technical considerations. As a result, the role of DDD, as well as what it does and perhaps more importantly doesn’t imply, become ostensibly clear. Many a time, my team and I would be at odds with the friction encountered in applying DDD. With *Implementing Domain-Driven Design* as our luminous guide we were able to overcome those challenges and translate our efforts into immediate busi

ness value.”

—Lev Gorodinski, Principal Architect, DrillSpot.com

From the Library of Javier Acero

**Implementing**

**Domain-Driven Design**

From the Library of Javier Acero

*This page intentionally left blank*

From the Library of Javier Acero

**Implementing**

**Domain-Driven**

**Design**

**Vaughn Vernon**

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco

New York • Toronto • Montreal • London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City

From the Library of Javier Acero

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liabil ity is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk pur chases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419

corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales

international@pearsoned.com

Visit us on the Web: informit.com/aw

*Library of Congress Control Number: 2012954071*

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited repro duction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-83457-7

ISBN-10: 0-321-83457-7

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts. First printing, January 2013

From the Library of Javier Acero

*This book is dedicated to my dearest Nicole and Tristan. Thanks for your love, your support, and your patience.*

From the Library of Javier Acero

*This page intentionally left blank*

From the Library of Javier Acero

**Contents**

**Foreword . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xvii Preface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xix Acknowledgments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxix About the Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxiii Guide to This Book . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxv**

**Chapter 1 Getting Started with DDD . . . . . . . . . . . . . . . . . . . 1** Can I DDD? . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2 Why You Should Do DDD . . . . . . . . . . . . . . . . . . . . . 6 How to Do DDD . . . . . . . . . . . . . . . . . . . . . . . . . . 20 The Business Value of Using DDD . . . . . . . . . . . . . . . . . 25 1. The Organization Gains a Useful Model of Its Domain . . . 26

2. A Refined, Precise Definition and Understanding of the

Business Is Developed . . . . . . . . . . . . . . . . . . . . 27

3. Domain Experts Contribute to Software Design . . . . . . . 27

4. A Better User Experience Is Gained . . . . . . . . . . . . . 27

5. Clean Boundaries Are Placed around Pure Models . . . . . . 28

6. Enterprise Architecture Is Better Organized . . . . . . . . . 28

7. Agile, Iterative, Continuous Modeling Is Used . . . . . . . . 28

8. New Tools, Both Strategic and Tactical, Are Employed . . . 28

The Challenges of Applying DDD . . . . . . . . . . . . . . . . . 29 Fiction, with Bucketfuls of Reality . . . . . . . . . . . . . . . . . 38 Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**ix**

From the Library of Javier Acero

**x** CONTENTS

**Chapter 2 Domains, Subdomains, and Bounded Contexts . . . . . . . . 43** Big Picture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

Subdomains and Bounded Contexts at Work . . . . . . . . . . 44

Focus on the Core Domain . . . . . . . . . . . . . . . . . . . 50

Why Strategic Design Is So Incredibly Essential . . . . . . . . . . 53

Real-World Domains and Subdomains . . . . . . . . . . . . . . . 56

Making Sense of Bounded Contexts . . . . . . . . . . . . . . . . 62

Room for More than the Model . . . . . . . . . . . . . . . . . 66

Size of Bounded Contexts . . . . . . . . . . . . . . . . . . . . 68

Aligning with Technical Components . . . . . . . . . . . . . 71

Sample Contexts . . . . . . . . . . . . . . . . . . . . . . . . . . 72

Collaboration Context. . . . . . . . . . . . . . . . . . . . . . 73

Identity and Access Context . . . . . . . . . . . . . . . . . . . 80

Agile Project Management Context . . . . . . . . . . . . . . . 82

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**Chapter 3 Context Maps . . . . . . . . . . . . . . . . . . . . . . . . . 87** Why Context Maps Are So Essential . . . . . . . . . . . . . . . 87

Drawing Context Maps . . . . . . . . . . . . . . . . . . . . . 89

Projects and Organizational Relationships . . . . . . . . . . . 91

Mapping the Three Contexts . . . . . . . . . . . . . . . . . . 95

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111

**Chapter 4 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 113** Interviewing the Successful CIO . . . . . . . . . . . . . . . . . 114

Layers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

Dependency Inversion Principle . . . . . . . . . . . . . . . . 123

Hexagonal or Ports and Adapters . . . . . . . . . . . . . . . . 125

Service-Oriented . . . . . . . . . . . . . . . . . . . . . . . . . 130

Representational State Transfer—REST . . . . . . . . . . . . . 133

REST as an Architectural Style . . . . . . . . . . . . . . . . 133

Key Aspects of a RESTful HTTP Server . . . . . . . . . . . 135

Key Aspects of a RESTful HTTP Client . . . . . . . . . . . 136

REST and DDD . . . . . . . . . . . . . . . . . . . . . . . . 136

Why REST? . . . . . . . . . . . . . . . . . . . . . . . . . . 138

From the Library of Javier Acero

CONTENTS **xi**

Command-Query Responsibility Segregation, or CQRS . . . . . 138 Examining Areas of CQRS . . . . . . . . . . . . . . . . . . 140

Dealing with an Eventually Consistent Query Model . . . . . 146

Event-Driven Architecture . . . . . . . . . . . . . . . . . . . . 147 Pipes and Filters . . . . . . . . . . . . . . . . . . . . . . . . 149

Long-Running Processes, aka Sagas . . . . . . . . . . . . . 153

Event Sourcing . . . . . . . . . . . . . . . . . . . . . . . . 160

Data Fabric and Grid-Based Distributed Computing . . . . . . 163 Data Replication . . . . . . . . . . . . . . . . . . . . . . . 164

Event-Driven Fabrics and Domain Events . . . . . . . . . . . 165

Continuous Queries . . . . . . . . . . . . . . . . . . . . . . 166

Distributed Processing . . . . . . . . . . . . . . . . . . . . 167

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168

**Chapter 5 Entities . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171** Why We Use Entities . . . . . . . . . . . . . . . . . . . . . . . 171 Unique Identity . . . . . . . . . . . . . . . . . . . . . . . . . . 173 User Provides Identity . . . . . . . . . . . . . . . . . . . . . 174

Application Generates Identity . . . . . . . . . . . . . . . . 175

Persistence Mechanism Generates Identity . . . . . . . . . . 179

Another Bounded Context Assigns Identity . . . . . . . . . . 182

When the Timing of Identity Generation Matters . . . . . . . 184

Surrogate Identity . . . . . . . . . . . . . . . . . . . . . . . 186

Identity Stability . . . . . . . . . . . . . . . . . . . . . . . . 188

Discovering Entities and Their Intrinsic Characteristics . . . . . 191 Uncovering Entities and Properties . . . . . . . . . . . . . . 192

Digging for Essential Behavior . . . . . . . . . . . . . . . . 196

Roles and Responsibilities . . . . . . . . . . . . . . . . . . 200

Construction . . . . . . . . . . . . . . . . . . . . . . . . . 205

Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . 208

Change Tracking . . . . . . . . . . . . . . . . . . . . . . . 216

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 217

**Chapter 6 Value Objects . . . . . . . . . . . . . . . . . . . . . . . . . 219** Value Characteristics . . . . . . . . . . . . . . . . . . . . . . . 221 Measures, Quantifies, or Describes . . . . . . . . . . . . . . 221

Immutable . . . . . . . . . . . . . . . . . . . . . . . . . . . 221

From the Library of Javier Acero

**xii** CONTENTS

Conceptual Whole . . . . . . . . . . . . . . . . . . . . . . 223

Replaceability . . . . . . . . . . . . . . . . . . . . . . . . . 226

Value Equality . . . . . . . . . . . . . . . . . . . . . . . . . 227

Side-Effect-Free Behavior . . . . . . . . . . . . . . . . . . . 228

Integrate with Minimalism . . . . . . . . . . . . . . . . . . . . 232

Standard Types Expressed as Values . . . . . . . . . . . . . . . 234

Testing Value Objects . . . . . . . . . . . . . . . . . . . . . . 239

Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . 243

Persisting Value Objects . . . . . . . . . . . . . . . . . . . . . 248

Reject Undue Influence of Data Model Leakage. . . . . . . . 249

ORM and Single Value Objects . . . . . . . . . . . . . . . . 251

ORM and Many Values Serialized into a Single Column . . . 253

ORM and Many Values Backed by a Database Entity. . . . . 255

ORM and Many Values Backed by a Join Table. . . . . . . . 260

ORM and Enum-as-State Objects . . . . . . . . . . . . . . . 261

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 263

**Chapter 7 Services . . . . . . . . . . . . . . . . . . . . . . . . . . . . 265** What a Domain Service Is (but First, What It Is Not) . . . . . . 267

Make Sure You Need a Service . . . . . . . . . . . . . . . . . . 268

Modeling a Service in the Domain . . . . . . . . . . . . . . . . 272

Is Separated Interface a Necessity? . . . . . . . . . . . . . . 275

A Calculation Process . . . . . . . . . . . . . . . . . . . . . 277

Transformation Services . . . . . . . . . . . . . . . . . . . . 280

Using a Mini-Layer of Domain Services . . . . . . . . . . . 281

Testing Services. . . . . . . . . . . . . . . . . . . . . . . . . . 281

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 284

**Chapter 8 Domain Events . . . . . . . . . . . . . . . . . . . . . . . . 285** The When and Why of Domain Events . . . . . . . . . . . . . 285

Modeling Events . . . . . . . . . . . . . . . . . . . . . . . . . 288

With Aggregate Characteristics . . . . . . . . . . . . . . . . 294

Identity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295

Publishing Events from the Domain Model . . . . . . . . . . . 296

Publisher . . . . . . . . . . . . . . . . . . . . . . . . . . . 297

Subscribers . . . . . . . . . . . . . . . . . . . . . . . . . . 300

From the Library of Javier Acero

CONTENTS **xiii**

Spreading the News to Remote Bounded Contexts . . . . . . . 303 Messaging Infrastructure Consistency . . . . . . . . . . . . 303

Autonomous Services and Systems . . . . . . . . . . . . . . 305

Latency Tolerances . . . . . . . . . . . . . . . . . . . . . . 306

Event Store . . . . . . . . . . . . . . . . . . . . . . . . . . . . 307 Architectural Styles for Forwarding Stored Events . . . . . . . . 312 Publishing Notifications as RESTful Resources . . . . . . . . 312

Publishing Notifications through Messaging Middleware . . 317

Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 318 Publishing the NotificationLog . . . . . . . . . . . . . . 319

Publishing Message-Based Notifications . . . . . . . . . . . 324

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 331

**Chapter 9 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . 333** Designing with Modules . . . . . . . . . . . . . . . . . . . . . 333 Basic Module Naming Conventions . . . . . . . . . . . . . . . 336 Module Naming Conventions for the Model . . . . . . . . . . . 337 Modules of the Agile Project Management Context . . . . . . . 340 Modules in Other Layers . . . . . . . . . . . . . . . . . . . . 343 Module before Bounded Context . . . . . . . . . . . . . . . . 344 Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 345

**Chapter 10 Aggregates . . . . . . . . . . . . . . . . . . . . . . . . . . 347** Using Aggregates in the Scrum Core Domain . . . . . . . . . . 348 First Attempt: Large-Cluster Aggregate . . . . . . . . . . . . 349

Second Attempt: Multiple Aggregates . . . . . . . . . . . . . 351

Rule: Model True Invariants in Consistency Boundaries . . . . . 353 Rule: Design Small Aggregates . . . . . . . . . . . . . . . . . . 355 Don’t Trust Every Use Case . . . . . . . . . . . . . . . . . . 358

Rule: Reference Other Aggregates by Identity . . . . . . . . . . 359 Making Aggregates Work Together through Identity

References . . . . . . . . . . . . . . . . . . . . . . . . . 361

Model Navigation . . . . . . . . . . . . . . . . . . . . . . . 362

Scalability and Distribution . . . . . . . . . . . . . . . . . . 363

Rule: Use Eventual Consistency Outside the Boundary . . . . . 364 Ask Whose Job It Is . . . . . . . . . . . . . . . . . . . . . . 366

From the Library of Javier Acero

**xiv** CONTENTS

Reasons to Break the Rules . . . . . . . . . . . . . . . . . . . 367

Reason One: User Interface Convenience . . . . . . . . . . . 367

Reason Two: Lack of Technical Mechanisms . . . . . . . . . 368

Reason Three: Global Transactions . . . . . . . . . . . . . . 369

Reason Four: Query Performance . . . . . . . . . . . . . . . 369

Adhering to the Rules . . . . . . . . . . . . . . . . . . . . . 370

Gaining Insight through Discovery. . . . . . . . . . . . . . . . 370

Rethinking the Design, Again . . . . . . . . . . . . . . . . . 370

Estimating Aggregate Cost . . . . . . . . . . . . . . . . . . 372

Common Usage Scenarios . . . . . . . . . . . . . . . . . . 373

Memory Consumption . . . . . . . . . . . . . . . . . . . . 374

Exploring Another Alternative Design . . . . . . . . . . . . 375

Implementing Eventual Consistency . . . . . . . . . . . . . . 376

Is It the Team Member’s Job? . . . . . . . . . . . . . . . . . 378

Time for Decisions . . . . . . . . . . . . . . . . . . . . . . 379

Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 380

Create a Root Entity with Unique Identity . . . . . . . . . . 380

Favor Value Object Parts . . . . . . . . . . . . . . . . . . . 382

Using Law of Demeter and Tell, Don’t Ask . . . . . . . . . . 382

Optimistic Concurrency . . . . . . . . . . . . . . . . . . . . 385

Avoid Dependency Injection . . . . . . . . . . . . . . . . . . 387

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 388

**Chapter 11 Factories . . . . . . . . . . . . . . . . . . . . . . . . . . . 389** Factories in the Domain Model . . . . . . . . . . . . . . . . . 389

Factory Method on Aggregate Root . . . . . . . . . . . . . . . 391

Creating CalendarEntry Instances . . . . . . . . . . . . . 392

Creating Discussion Instances . . . . . . . . . . . . . . . 395

Factory on Service . . . . . . . . . . . . . . . . . . . . . . . . 397

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 400

**Chapter 12 Repositories . . . . . . . . . . . . . . . . . . . . . . . . . 401** Collection-Oriented Repositories . . . . . . . . . . . . . . . . 402

Hibernate Implementation . . . . . . . . . . . . . . . . . . 407

Considerations for a TopLink Implementation . . . . . . . . 416

From the Library of Javier Acero

CONTENTS **xv**

Persistence-Oriented Repositories . . . . . . . . . . . . . . . . 418 Coherence Implementation . . . . . . . . . . . . . . . . . . 420

MongoDB Implementation . . . . . . . . . . . . . . . . . . 425

Additional Behavior . . . . . . . . . . . . . . . . . . . . . . . 430 Managing Transactions . . . . . . . . . . . . . . . . . . . . . 432 A Warning . . . . . . . . . . . . . . . . . . . . . . . . . . 437

Type Hierarchies . . . . . . . . . . . . . . . . . . . . . . . . . 437 Repository versus Data Access Object . . . . . . . . . . . . . . 440 Testing Repositories . . . . . . . . . . . . . . . . . . . . . . . 441 Testing with In-Memory Implementations . . . . . . . . . . 445

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 448

**Chapter 13 Integrating Bounded Contexts . . . . . . . . . . . . . . . . 449** Integration Basics . . . . . . . . . . . . . . . . . . . . . . . . 450 Distributed Systems Are Fundamentally Different . . . . . . 451

Exchanging Information across System Boundaries . . . . . . 452

Integration Using RESTful Resources . . . . . . . . . . . . . . 458 Implementing the RESTful Resource . . . . . . . . . . . . . 459

Implementing the REST Client Using an Anticorruption

Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . 463

Integration Using Messaging . . . . . . . . . . . . . . . . . . . 469 Staying Informed about Product Owners and Team

Members . . . . . . . . . . . . . . . . . . . . . . . . . . 469

Can You Handle the Responsibility? . . . . . . . . . . . . . 476

Long-Running Processes, and Avoiding Responsibility . . . . 481

Process State Machines and Time-out Trackers . . . . . . . . 493

Designing a More Sophisticated Process . . . . . . . . . . . 503

When Messaging or Your System Is Unavailable . . . . . . . 507

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 508

**Chapter 14 Application. . . . . . . . . . . . . . . . . . . . . . . . . . 509** User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 512 Rendering Domain Objects . . . . . . . . . . . . . . . . . . 512

Render Data Transfer Object from Aggregate Instances . . . 513

Use a Mediator to Publish Aggregate Internal State . . . . . . 514

Render Aggregate Instances from a Domain Payload Object 515

From the Library of Javier Acero

**xvi** CONTENTS

State Representations of Aggregate Instances . . . . . . . . . 516

Use Case Optimal Repository Queries. . . . . . . . . . . . . 517

Dealing with Multiple, Disparate Clients . . . . . . . . . . . 517

Rendition Adapters and Handling User Edits . . . . . . . . . 518

Application Services . . . . . . . . . . . . . . . . . . . . . . . 521

Sample Application Service . . . . . . . . . . . . . . . . . . 522

Decoupled Service Output . . . . . . . . . . . . . . . . . . 528

Composing Multiple Bounded Contexts . . . . . . . . . . . . . 531

Infrastructure . . . . . . . . . . . . . . . . . . . . . . . . . . 532

Enterprise Component Containers . . . . . . . . . . . . . . . . 534

Wrap-Up . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 537

**Appendix A Aggregates and Event Sourcing: A+ES . . . . . . . . . . . 539** Inside an Application Service . . . . . . . . . . . . . . . . . . 541

Command Handlers . . . . . . . . . . . . . . . . . . . . . . . 549

Lambda Syntax . . . . . . . . . . . . . . . . . . . . . . . . . 553

Concurrency Control. . . . . . . . . . . . . . . . . . . . . . . 554

Structural Freedom with A+ES . . . . . . . . . . . . . . . . . . 558

Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . 558

Implementing an Event Store . . . . . . . . . . . . . . . . . . . 561

Relational Persistence . . . . . . . . . . . . . . . . . . . . . . 565

BLOB Persistence . . . . . . . . . . . . . . . . . . . . . . . . 568

Focused Aggregates . . . . . . . . . . . . . . . . . . . . . . . 569

Read Model Projections . . . . . . . . . . . . . . . . . . . . . 570

Use with Aggregate Design . . . . . . . . . . . . . . . . . . . 573

Events Enrichment . . . . . . . . . . . . . . . . . . . . . . . . 573

Supporting Tools and Patterns . . . . . . . . . . . . . . . . . . 576

Event Serializers . . . . . . . . . . . . . . . . . . . . . . . . 576

Event Immutability . . . . . . . . . . . . . . . . . . . . . . 577

Value Objects . . . . . . . . . . . . . . . . . . . . . . . . . 577

Contract Generation . . . . . . . . . . . . . . . . . . . . . . . 580

Unit Testing and Specifications . . . . . . . . . . . . . . . . . 582

Event Sourcing in Functional Languages . . . . . . . . . . . . . 583

**Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 585 Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 589**

From the Library of Javier Acero

**Foreword**

In this new book, Vaughn Vernon presents the whole of Domain-Driven Design (DDD) in a distinctive way, with new explanations of the concepts, new examples, and an original organization of topics. I believe this fresh, alter native approach will help people grasp the subtleties of DDD, particularly the more abstract ones such as Aggregates and Bounded Contexts. Not only do different people prefer different styles—subtle abstractions are hard to absorb without multiple explanations.

Also, the book conveys some of the insights of the past nine years that have been described in papers and presentations but have not appeared in a book before now. It places Domain Events alongside Entities and Value Objects as the building blocks of a model. It discusses the Big Ball of Mud and places it into the Context Map. It explains the hexagonal architecture, which has emerged as a better description of what we do than the layered architecture.

My first exposure to the material in this book came almost two years ago (although Vaughn had been working on his book for some time by then). At the first DDD Summit, several of us committed to writing about certain topics about which we felt there were fresh things to say or there was a particular need in the community for more specific advice. Vaughn took up the challenge of writing about Aggregates, and he followed through with a series of excellent articles about Aggregates (which became a chapter in this book).

There was also a consensus at the summit that many practitioners would benefit from a more prescriptive treatment of some of the DDD patterns. The honest answer to almost any question in software development is, “It depends.” That is not very useful to people who want to learn to apply a tech

nique, however. A person who is assimilating a new subject needs concrete guidance. Rules of thumb don’t have to be right in all cases. They are what usually works well or the thing to try first. Through their decisiveness, they convey the philosophy of the approach to solving the problem. Vaughn’s book has a good mix of straightforward advice balanced with a discussion of trade

offs that keep it from being simplistic.

**xvii**

From the Library of Javier Acero

**xviii** FOREWORD

Not only have additional patterns, such as Domain Events, become a main

stream part of DDD—people in the field have progressed in learning how to apply those patterns, not to mention adapting them to newer architectures and technologies. Nine years after my book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, was published, there’s actually a lot to say about DDD that is new, and there are new ways to talk about the funda

mentals. Vaughn’s book is the most complete explanation yet of those new insights into practicing DDD.

—Eric Evans

Domain Language, Inc.

From the Library of Javier Acero

**Preface**

*All the calculations show it can’t work. There’s only one thing to do: make it work.*

*—Pierre-Georges Latécoère,*

*early French aviation entrepreneur*

And make it work we shall. The Domain-Driven Design approach to software development is far too important to leave any capable developer without clear directions for how to implement it successfully.

**Getting Grounded, Getting Airborne**

When I was a kid, my father learned to pilot small airplanes. Often the whole family would go up flying. Sometimes we flew to another airport for lunch, then returned. When Dad had less time but longed to be in the air, we’d go out, just the two of us, and circle the airport doing “touch-and-goes.”

We also took some long trips. For those, we always had a map of the route that Dad had earlier charted. Our job as kids was to help navigate by looking out for landmarks below so we could be certain to stay on course. This was great fun for us because it was a challenge to spot objects so far below that exhibited little in the way of identifying details. Actually, I’m sure that Dad always knew where we were. He had all the instruments on the dashboard, and he was licensed for instrument flight.

The view from the air really changed my perspective. Now and then Dad and I would fly over our house in the countryside. At a few hundred feet up, this gave me a context for home that I didn’t have before. As Dad would cruise over our house, Mom and my sisters would run out into the yard to wave at us. I knew it was them, although I couldn’t look into their eyes. We couldn’t

**xix**

From the Library of Javier Acero

**xx** PREFACE

converse. If I had shouted out the airplane window, they would never have heard me. I could see the split-rail fence in the front dividing our property from the road. When on the ground I’d walk across it as if on a balance beam. From the air, it looked like carefully woven twigs. And there was the huge yard that I circled row by row on our riding lawn mower every summer. From the air, I saw only a sea of green, not the blades of grass.

I loved those moments in the air. They are etched in my memory as if Dad

and I were just taxiing in after landing to tie down for the evening. As much as I loved those flights, they sure were no substitute for being on the ground. And as cool as they were, the touch-and-goes were just too brief to make me feel grounded.

**Landing with Domain-Driven Design**

Getting in touch with Domain-Driven Design (DDD) can be like flight to a kid. The view from the air is stunning, but sometimes things look unfamiliar enough to prevent us from knowing exactly where we are. Getting from point A to point B appears far from realistic. The DDD grownups always seem to know where they are. They’ve long ago plotted a course, and they are com

pletely in tune with their navigational instruments. A great number of oth ers don’t feel grounded. What is needed is the ability to “land and tie down.” Next, a map is needed to guide the way from where we are to where we need to be.

In the book *Domain-Driven Design: Tackling Complexity in the Heart of*

*Software* [Evans], Eric Evans brought about what is a timeless work. It is my firm belief that Eric’s work will guide developers in practical ways for decades to come. Like other pattern works, it establishes flight far enough above the surface to give a broad vision. Yet, there may be a bit more of a challenge when we need to understand the groundwork involved in implementing DDD, and we usually desire more detailed examples. If only we could land and stay on the surface a bit longer, and even drive home or to some other familiar place.

Part of my goal is to take you in for a soft landing, secure the aircraft, and

help you get home by way of a well-known surface route. That will help you make sense of implementing DDD, giving you examples that use familiar tools and technologies. And since none of us can stay home all the time, I will also help you venture out onto other paths to explore new terrain, taking you to places that perhaps you’ve never been before. Sometimes the path will be steep, but given the right tactics, a challenging yet safe ascent is possible. On this trip you’ll learn about alternative architectures and patterns for integrating

From the Library of Javier Acero

MAPPING THE TERRAIN AND CHARTING FOR FLIGHT **xxi**

multiple domain models. This may expose you to some previously unexplored territory. You will find detailed coverage of strategic modeling with multiple integrations, and you’ll even learn how to develop autonomous services.

My goal is to provide a map to help you take both short jaunts and long, complicated treks, enjoying the surrounding detail, without getting lost or injured along the way.

**Mapping the Terrain and Charting for Flight**

It seems that in software development we are always mapping from one thing to another. We map our objects to databases. We map our objects to the user interface and then back again. We map our objects to and from various appli cation representations, including those that can be consumed by other systems and applications. With all this mapping, it’s natural to want a map from the higher-level patterns of Evans to implementation.

Even if you have already landed a few times with DDD, there is probably more to benefit from. Sometimes DDD is first embraced as a technical tool set. Some refer to this approach to DDD as *DDD-Lite.* We may have homed in on Entities, Services, possibly made a brave attempt at designing Aggregates, and tried to manage their persistence using Repositories. Those patterns felt a bit like familiar ground, so we put them to use. We may even have found some use for Value Objects along the way. All of these fall within the catalog of *tacti*

*cal design* patterns, which are more technical. They help us take on a serious software problem with the skill of a surgeon with a scalpel. Still, there is much to learn about these and other places to go with tactical design as well. I map them to implementation.

Have you traveled beyond tactical modeling? Have you visited and even lin gered with what some call the “other half” of DDD, the *strategic design* pat terns? If you’ve left out the use of Bounded Context and Context Maps, you have probably also missed out on the use of the Ubiquitous Language.

If there is a single “invention” Evans delivers to the software development community, it is the Ubiquitous Language. At a minimum he brought the Ubiq uitous Language out of the dusty archives of design wisdom. It is a team pat tern used to capture the concepts and terms of a specific core business domain in the software model itself. The software model incorporates the nouns, adjec tives, verbs, and richer expressions formally spoken by the development team, a team that includes one or more business domain experts. It would be a mis take, however, to conclude that the Language is limited to mere words. Just as any human language reflects the minds of those who speak it, the Ubiquitous

From the Library of Javier Acero

**xxii** PREFACE

Language reflects the mental model of the experts of the business domain you are working in. Thus, the software and the tests that verify the model’s adher ence to the tenets of the domain both capture and adhere to this Language, the same conceived and spoken by the team. The Language is equally as valuable as the various strategic and tactical modeling patterns and in some cases has a more enduring quality.

Simply stated, practicing DDD-Lite leads to the construction of inferior

domain models. That’s because the Ubiquitous Language, Bounded Context, and Context Mapping have so much to offer. You get more than a team lingo. The Language of a team in an explicit Bounded Context expressed as a domain model adds true business value and gives us certainty that we are implement

ing the correct software. Even from a technical standpoint, it helps us create better models, ones with more potent behaviors, that are pure and less error prone. Thus, I map the strategic design patterns to understandable example implementations.

This book maps the terrain of DDD in a way that allows you to experience

the benefits of both strategic and tactical design. It puts you in touch with its business value and technical strengths by peering closely at the details.

It would be a disappointment if all we ever did with DDD is stay on the

ground. Getting stuck in the details, we’d forget that the view from flight teaches us a lot, too. Don’t limit yourself to rugged ground travel. Brave the challenge of getting in the pilot’s seat and see from a height that is telling. With training flights on strategic design, with its Bounded Contexts and Context Maps, you will be prepared to gain a grander perspective on its full realization. When you reward yourself with DDD flight, I will have reached my goal.

**Summary of Chapters**

The following highlights the chapters of this book and how you can benefit from each one.

**Chapter 1: Getting Started with DDD**

This chapter introduces you to the benefits of using DDD and how to achieve the most from it. You will learn what DDD can do for your projects and your teams as you grapple with complexity. You’ll find out how to score your proj ect to see if it deserves the DDD investment. You will consider the common alternatives to DDD and why they often lead to problems. The chapter lays the foundations of DDD as you learn how to take the first steps on your project,

From the Library of Javier Acero

SUMMARY OF CHAPTERS **xxiii**

and it even gives you some ways to sell DDD to your management, domain experts, and technical team members. That will enable you to face the chal lenges of using DDD armed with the knowledge of how to succeed.

You are introduced to a project case study that involves a fictitious company and team, yet one with real-world DDD challenges. The company, with the charter to create innovative SaaS-based products in a multitenant environment, experiences many of the mistakes common to DDD adoption but makes vital discoveries that help the teams solve their issues and keep the project on track. The project is one that most developers can relate to, as it involves developing a Scrum-based project management application. This case study introduction sets the stage for subsequent chapters. Each strategic and tactical pattern is taught through the eyes of the team, both as they err and as they make strides toward maturity in implementing DDD successfully.

**Chapter 2: Domains, Subdomains, and Bounded Contexts**

What is a Domain, a Subdomain, and a Core Domain? What are Bounded Con texts, and why and how should you use them? These questions are answered in the light of mistakes made by the project team in our case study. Early on in their first DDD project they failed to understand the Subdomain they were working within, its Bounded Context, and a concise Ubiquitous Language. In fact, they were completely unfamiliar with strategic design, only leveraging the tactical patterns for their technical benefits. This led to problems in their ini tial domain model design. Fortunately, they recognized what had happened before it became a hopeless morass.

A vital message is conveyed, that of applying Bounded Contexts to distin guish and segregate models properly. Addressed are common misapplications of the pattern along with effective implementation advice. The text then leads you through the corrective steps the team took and how that resulted in the creation of two distinct Bounded Contexts. This led to the proper separation of modeling concepts in their third Bounded Context, the new Core Domain, and the main sample used in the book.

This chapter will strongly resonate with readers who have felt the pain of applying DDD only in a technical way. If you are uninitiated in strategic design, you are pointed in the right direction to start out on a successful journey.

**Chapter 3: Context Maps**

Context Maps are a powerful tool to help a team understand their business domain, the boundaries between distinct models, and how they are currently, or can be, integrated. This technique is not limited to drawing a diagram of

From the Library of Javier Acero

**xxiv** PREFACE

your system architecture. It’s about understanding the relationships between the various Bounded Contexts in an enterprise and the patterns used to map objects cleanly from one model to another. Use of this tool is important to suc ceeding with Bounded Contexts in a complex business enterprise. This chapter takes you through the process used by the project team as they applied Context Mapping to understand the problems they created with their first Bounded Context (Chapter 2). It then shows how the two resulting clean Bounded Con texts were leveraged by the team responsible for designing and implementing the new Core Domain.

**Chapter 4: Architecture**

Just about everyone knows the Layers Architecture. Are Layers the only way to house a DDD application, or can other diverse architectures be used? Here we consider how to use DDD within such architectures as Hexagonal (Ports and Adapters), Service-Oriented, REST, CQRS, Event-Driven (Pipes and Filters, Long-Running Processes or Sagas, Event Sourcing), and Data Fabric/Grid

Based. Several of these architectural styles were put to use by the project team.

**Chapter 5: Entities**

The first of the DDD tactical patterns treated is Entities. The project team first leaned too heavily on these, overlooking the importance of designing with Value Objects when appropriate. This led to a discussion of how to avoid wide spread overuse of Entities because of the undue influence of databases and per sistence frameworks.

Once you are familiar with ways to distinguish their proper use, you see

lots of examples of how to design Entities well. How do we express the Ubiq uitous Language with an Entity? How are Entities tested, implemented, and persisted? You are stepped through how-to guidance for each of these.

**Chapter 6: Value Objects**

Early on the project team missed out on important modeling opportunities with Value Objects. They focused too intensely on the individual attributes of Entities when they should have been giving careful consideration to how multiple related attributes are properly gathered as an immutable whole. This chapter looks at Value Object design from several angles, discussing how to identify the special characteristics in the model as a means to determine when to use a Value rather than an Entity. Other important topics are covered, such as the role of Values in integration and modeling Standard Types. The chapter then shows how to design domain-centric tests, how to implement Value types,

From the Library of Javier Acero

SUMMARY OF CHAPTERS **xxv**

and how to avoid the bad influence persistence mechanisms can have on our need to store them as part of an Aggregate.

**Chapter 7: Services**

This chapter shows how to determine when to model a concept as a fine grained, stateless Service that lives in the domain model. You are shown when you should design a Service instead of an Entity or Value Object, and how Domain Services can be implemented to handle business domain logic as well as for technical integration purposes. The decisions of the project team are used to exemplify when to use Services and how they are designed.

**Chapter 8: Domain Events**

Domain Events were not formally introduced by Eric Evans as part of DDD until after his book was published. You’ll learn why Domain Events published by the model are so powerful, and the diverse ways that they can be used, even in supporting integration and autonomous business services. Although various kinds of technical events are sent and processed by applications, the distinguishing characteristics of Domain Events are spotlighted. Design and implementation guidance is provided, instructing you on available options and trade-offs. The chapter then teaches how to create a Publish-Subscribe mech

anism, how Domain Events are published to integrated subscribers across the enterprise, ways to create and manage an Event Store, and how to properly deal with common messaging challenges faced. Each of these areas is discussed in light of the project team’s efforts to use them correctly and to their best advantage.

**Chapter 9: Modules**

How do we organize model objects into right-sized containers with limited coupling to objects that are in different containers? How do we name these containers so they reflect the Ubiquitous Language? Beyond packages and namespaces, how can we use the more modern modularization facilities, such as OSGi and Jigsaw, provided by languages and frameworks? Here you will see how Modules were put to use by the project team across a few of their projects.

**Chapter 10: Aggregates**

Aggregates are probably the least well understood among DDD’s tactical tools. Yet, if we apply some rules of thumb, Aggregates can be made simpler and quicker to implement. You will learn how to cut through the complexity

From the Library of Javier Acero

**xxvi** PREFACE

barrier to use Aggregates that create consistency boundaries around small object clusters. Because of putting too much emphasis on the less important aspects of Aggregates, the project team in our case study stumbled in a few different ways. We step through the team’s iterations with a few modeling chal

lenges and analyze what went wrong and what they did about it. The result of their efforts led to a deeper understanding of their Core Domain. We look in on how the team corrected their mistakes through the proper application of transactional and eventual consistency, and how that led them to design a more scalable and high-performing model within a distributed processing environment.

**Chapter 11: Factories**

[Gamma et al.] has plenty to say about Factories, so why bother with treating them in this book? This is a simple chapter that does not attempt to reinvent the wheel. Rather, its focus is on understanding *where* Factories should exist. There are, of course, a few good tips to share about designing a worthy Fac

tory in a DDD setting. See how the project team created Factories in their Core Domain as a way to simplify the client interface and protect the model’s con sumers from introducing disastrous bugs into their multitenant environment.

**Chapter 12: Repositories**

Isn’t a Repository just a simple Data Access Object (DAO)? If not, what’s the difference? Why should we consider designing Repositories to mimic collec tions rather than databases? Learn how to design a Repository that is used with an ORM, one that supports the Coherence grid-based distributed cache, and one that uses a NoSQL key-value store. Each of these optional persistence mechanisms was at the disposal of the project team because of the power and versatility behind the Repository building block pattern.

**Chapter 13: Integrating Bounded Contexts**

Now that you understand the higher-level techniques of Context Mapping and have the tactical patterns on your side, what is involved in actually implement ing the integrations between models? What integration options are afforded by DDD? This chapter uncovers a few different ways to implement model inte grations using Context Mapping. Instruction is given based on how the project team integrated the Core Domain with other supporting Bounded Contexts introduced in early chapters.

From the Library of Javier Acero

JAVA AND DEVELOPMENT TOOLS **xxvii**

**Chapter 14: Application**

You have designed a model per your Core Domain’s Ubiquitous Language. You’ve developed ample tests around its usage and correctness, and it works. But how do other members of your team design the areas of the application that surround the model? Should they use DTOs to transfer data between the model and the user interface? Or are there other options for conveying model state up to the presentation components? How do the Application Services and infrastructure work? This chapter addresses those concerns using the now familiar project to convey available options.

**Appendix A: Aggregates and Event Sourcing: A+ES**

Event Sourcing is an important technical approach to persisting Aggregates that also provides the basis for developing an Event-Driven Architecture. Event Sourcing can be used to represent the entire state of an Aggregate as a sequence of Events that have occurred since it was created. The Events are used to rebuild the state of the Aggregate by replaying them in the same order in which they occurred. The premise is that this approach simplifies persistence and allows capturing concepts with complex behavioral properties, besides the far-reaching influence the Events themselves can have on your own and exter

nal systems.

**Java and Development Tools**

The majority of the examples in this book use the Java Programming Lan guage. I could have provided the examples in C#, but I made a conscious deci sion to use Java instead.

First of all, and sad to say, I think there has been a general abandonment of good design and development practices in the Java community. These days it may be difficult to find a clean, explicit domain model in most Java-based projects. It seems to me that Scrum and other agile techniques are being used as substitutes for careful modeling, where a product backlog is thrust at devel

opers as if it serves as a set of designs. Most agile practitioners will leave their daily stand-up without giving a second thought to how their backlog tasks will affect the underlying model of the business. Although I assume this is needless to say, I must assert that Scrum, for example, was never meant to stand in place of design. No matter how many project and product managers would like to keep you marching on a relentless path of continuous delivery, Scrum

From the Library of Javier Acero

**xxviii** PREFACE

was not meant only as a means to keep Gantt chart enthusiasts happy. Yet, it has become that in so many cases.

I consider this a big problem, and a major theme I have is to inspire the Java

community to return to domain modeling by giving a reasonable amount of thought to how sound, yet agile and rapid, design techniques can benefit their work.

Further, there are already some good resources for using DDD in a .NET

environment, one being *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* by Jimmy Nilsson [Nilsson]. Due to Jimmy’s good work and that of others promoting the Alt.NET mindset, there is a high tide of good design and development practices going on in the .NET community. Java developers need to take notice.

Second, I am well aware that the C#.NET community will have no problem

whatsoever understanding Java code. Due to the fact that much of the DDD community uses C#.NET, most of my early book reviewers are C# developers, and I never once received a complaint about their having to read Java code. So, I have no concern that my use of Java in any way alienates C# developers.

I need to add that at the time of this writing there was a significant shift

toward interest in using document-based and key-value storage over rela tional databases. This is for good reason, for even Martin Fowler has aptly nicknamed these “aggregate-oriented storage.” It’s a fitting name and well describes the advantages of using NoSQL storage in a DDD setting.

Yet, in my consulting work I find that many are still quite married to

relational databases and object-relational mapping. Therefore, I think that in practical terms there has been no disservice to the community of NoSQL enthusiasts by my including guidance on using object-relational mapping tech niques for domain models. I do acknowledge, however, that this may earn me some scorn from those who think that the object-relational impedance mis match makes it unworthy of consideration. That’s fine, and I accept the flames, because there is a vast majority who must still live with the drudgeries of this impedance mismatch on a day-to-day basis, however unenlightened they may seem to the minority.

Of course, I also provide guidance in Chapter 12, “Repositories,” on the use

of document-based, key-value, and Data Fabric/Grid-Based stores. As well, in several places I discuss where the use of a NoSQL store would tend to influence an alternative design of Aggregates and their contained parts. It’s quite likely that the trend toward NoSQL stores will continue to spur growth in that sec

tor, so in this case object-relational developers need to take notice. As you can see, I understand both sides of the argument, and I agree with both. It’s all part of the ongoing friction created by technology trends, and the friction needs to happen in order for positive change to happen.

From the Library of Javier Acero

**Acknowledgments**

I am grateful to the fine staff at Addison-Wesley for giving me the opportu nity to publish under their highly respected label. As I have stated before in my classes and presentations, I see Addison-Wesley as a publisher that under stands the value of DDD. Both Christopher Guzikowski and Chris Zahn (Dr. Z) have supported my efforts throughout the editorial process. I will not forget the day that Christopher Guzikowski called to share the news that he wanted to sign me as one of his authors. I will remember how he encouraged me to persevere through the doubts that most authors must experience, until publication was in sight. Of course, it was Dr. Z who made sure the text was put into a publishable state. Thanks to my production editor, Elizabeth Ryan, for coordinating the book’s publication details. And thanks to my intrepid copyeditor, Barbara Wood.

Going back a ways, it was Eric Evans who devoted a major portion of five years of his career to write the first definitive work on DDD. Without his efforts, the wisdom that grew out of the Smalltalk and patterns communities, and that Eric himself refined, many more developers would just be hacking their way to delivering bad software. Sadly, this problem is more common than it should be. As Eric says, the poor quality of software development, and the uncreative joylessness of the teams that produce the software, nearly drove him to exit the software industry for good. We owe Eric hearty thanks for concen

trating his energy into educating rather than into a career change.

At the end of the first DDD Summit in 2011, which Eric invited me to attend, it was determined that the leadership should produce a set of guidelines by which more developers could succeed with DDD. I was already far along with this book and was in a good position to understand what developers were missing. I offered to write an essay to provide the “rules of thumb” for Aggre

gates. I determined that this three-part series entitled “Effective Aggregate Design” would form the foundation for Chapter 10 of this book. Once released on dddcommunity.org, it became quite clear how such sound guidance was

**xxix**

From the Library of Javier Acero

**xxx** ACKNOWLEDGMENTS

greatly needed. Thanks to others among the DDD leadership who reviewed that essay and thus provided valuable feedback for this book. Eric Evans and Paul Rayner did several detailed reviews of the essay. I also received feedback from Udi Dahan, Greg Young, Jimmy Nilsson, Niclas Hedhman, and Rickard Öberg.

Special thanks go to Randy Stafford, a longtime member of the DDD com

munity. After attending a DDD talk I gave several years ago in Denver, Randy urged me to become more involved in the larger DDD community. Sometime later, Randy introduced me to Eric Evans so I could pitch my ideas about draw ing the DDD community together. While my ideas were a bit grander and possibly less achievable, Eric convinced us that forming a smaller contingent composed of clear DDD leadership would have more near-term value. From these discussions the DDD Summit 2011 was formed. Needless to say, without Randy’s coaxing me to push forward with my views of DDD, this book would not exist, and perhaps not even a DDD Summit. Although Randy was too busy with Oracle Coherence work to contribute to this book, perhaps we will get the chance to write something in the future in a combined effort.

A huge thank-you goes to Rinat Abdullin, Stefan Tilkov, and Wes Williams

for contributing sections about specialized topics to the text. It’s nearly impossi ble to know everything about everything related to DDD, and absolutely impos sible to be an expert in all areas of software development. That’s why I turned to experts in specific areas to write a few sections of Chapter 4 and Appendix A. Thanks go to Stefan Tilkov for his uncommon knowledge of REST, to Wes Wil liams for his GemFire experience, and to Rinat Abdullin for sharing his contin ually expanding experience with Event Sourcing for Aggregate implementation. One of my earliest reviewers was Leo Gorodinsk, and he stuck with the

project. I first met Leo at our DDD Denver meetup. He provided a lot of great feedback on this book based on his own struggles while implementing DDD with his team in Boulder, Colorado. I hope my book helped Leo as much as his critical reviews helped me. I see Leo as part of DDD’s future.

Many others provided feedback on at least one chapter of my book, and

some on several chapters. Some of the more critical feedback was provided by Gojko Adzic, Alberto Brandolini, Udi Dahan, Dan Haywood, Dave Muir head, and Stefan Tilkov. Specifically, Dan Haywood and Gojko Adzic deliv ered much of the early feedback, which was based on the most-painful-to-read content I produced. I am glad they endured and corrected me. Alberto Bran dolini’s insights into strategic design in general, and Context Mapping specif ically, helped me focus on the essence of that vital material. Dave Muirhead, with an abundance of experience in object-oriented design, domain modeling, as well as object persistence and in-memory data grids—including GemFire

From the Library of Javier Acero

ACKNOWLEDGMENTS **xxxi**

and Coherence—influenced my text regarding some of the history and finer details of object persistence. Besides his REST contribution, Stefan Tilkov sup plied additional insights into architecture in general, and SOA and Pipes and Filters specifically. Finally, Udi Dahan validated and helped me clarify some of the concepts of CQRS, Long-Running Processes (aka Sagas), and messag ing with NServiceBus. Other reviewers who provided valuable feedback were Rinat Abdullin, Svein Arne Ackenhausen, Javier Ruiz Aranguren, William Doman, Chuck Durfee, Craig Hoff, Aeden Jameson, Jiwei Wu, Josh Maletz, Tom Marrs, Michael McCarthy, Rob Meidal, Jon Slenk, Aaron Stockton, Tom Stockton, Chris Sutton, and Wes Williams.

Scorpio Steele produced the fantastic illustrations for the book. Scorpio made everyone on the IDDD team the superheroes that they truly are. At the other end of the spectrum was the nontechnical editorial review by my good friend Kerry Gilbert. While everyone else made sure I was technically correct, Kerry put me “under the grammar hammer.”

My father and mother have provided great inspiration and support through out my life. My father—AJ in the “Cowboy Logic” humor throughout this book—is not *just* a cowboy. Don’t get me wrong. Being a great cowboy would be enough. Besides loving flight and piloting airplanes, my father was an accomplished civil engineer and land surveyor, and a talented negotiator. He still loves math and studying the galaxies. Among many other things he taught me, my Dad imparted to me how to solve a right triangle when I was around ten years old. Thanks, Dad, for giving me a technical bent at a young age. Thanks also go to my mom, one of the nicest people you could ever know. She has always encouraged and supported me through my personal challenges. Besides, what stamina I have comes from her. I could go on, but I could never say enough good things about her.

Although this book is dedicated to my loving wife, Nicole, and our marvel ous son, Tristan, my thanks would not be complete without a special mention here. They are the ones who allowed me to work on and complete the book. Without their support and encouragement my task would not have been possi ble. Thanks so much, my dearest loved ones.

From the Library of Javier Acero

*This page intentionally left blank*

From the Library of Javier Acero

**About the Author**

**Vaughn Vernon** is a veteran software craftsman with more than twenty-five years of experience in software design, development, and architecture. He is a thought leader in simplifying software design and implementation using innovative meth ods. He has been programming with object-oriented languages since the 1980s and applying the tenets of Domain-Driven Design since his Smalltalk domain modeling days in the early 1990s. His experience spans a wide range of business domains, including aerospace, environmental, geospatial, insurance, medical and health care, and telecommunications. He has also succeeded in technical endeav ors, creating reusable frameworks, libraries, and implementation acceleration tools. He consults and speaks internationally and has taught his Implementing Domain-Driven Design classes on multiple continents. You can read more about his latest efforts at www.VaughnVernon.co and follow him on Twitter here: @VaughnVernon.

**xxxiii**

From the Library of Javier Acero

*This page intentionally left blank*

From the Library of Javier Acero

**Guide to This Book**

The book *Domain-Driven Design* by Eric Evans presents what is essentially a large *pattern language.* A pattern language is a set of software patterns that are intertwined because they are dependent on each other. Any one pattern references one or more other patterns that it depends on, or that depend on it. What does this mean for you?

It means that as you read any given chapter of this book, you could run into a DDD pattern that isn’t discussed in that chapter and that you don’t already know. Don’t panic, and please don’t stop reading out of frustration. The refer enced pattern is very likely explained in detail in another chapter of the book.

In order to help unravel the pattern language, I used the syntax found in Table G.1 in the text.

**Table G.1** The Syntax Used in This Book

**When You See This . . . It Means This . . .**

**Pattern Name (#)** 1. It is the first time the pattern is referenced in the chapter that you are reading, or

2. It is an important additional reference to a pattern

that was already mentioned in the chapter, but it’s

essential to know where to locate more information

about it at that point in the text.

**Bounded Context (2)** The chapter you are reading is referencing Chapter 2 for you to find out deep details about Bounded

Contexts.

Bounded Context It is the way I reference a pattern already mentioned in the same chapter. I don’t want to irritate you by

making every reference to a given pattern bold, with a

chapter number.

[REFERENCE] It is a bibliographic reference to another work.

*continues*

**xxxv**

From the Library of Javier Acero

**xxxvi** GUIDE TO THIS BOOK

**Table G.1** The Syntax Used in This Book (*Continued* )

**When You See This . . . It Means This . . .**

[Evans] or [Evans, Ref] I don’t cover the specific referenced DDD pattern extensively, and if you want to know more, you need

to read these works by Eric Evans. (They’re always

recommended reading!)

[Evans] means his classic book, *Domain-Driven*

*Design.*

[Evans, Ref] means a second publication that is a

separate, condensed reference to the patterns in [Evans]

that have been updated and extended.

[Gamma et al.] and [Fowler, P of EAA]

[Gamma et al.] means the classic book *Design Patterns.*

[Fowler, P of EAA] means Martin Fowler’s *Patterns of Enterprise Application Architecture.*

I reference these works frequently. Although I reference several other works as well, you will tend to see these a bit more than others. Examine the full bibliography for details.

If you start reading in the middle of a chapter and you see a reference such as Bounded Context, remember that you’ll probably find a chapter in this book that covers the pattern. Just glance at the index for a richer set of references.

If you have already read [Evans] and you know its patterns to some degree, you’ll probably tend to use this book as a way to clarify your understanding of DDD and to get ideas for how to improve your existing model designs. In that case you may not need a big-picture view right now. But if you are rela

tively new to DDD, the following section will help you see how the patterns fit together, and how this book can be used to get you up and running quickly. So, read on.

**Big-Picture View of DDD**

Early on I take you through one of the pillars of DDD, the **Ubiquitous Lan guage (1)**. A Ubiquitous Language is applicable within a single **Bounded Con text (2)**. Straightaway, you need to familiarize yourself with that critical domain modeling mindset. Just remember that whichever way your software models are designed *tactically*, *strategically* you’ll want them to reflect the following: a clean Ubiquitous Language modeled in an explicitly Bounded Context.

From the Library of Javier Acero

GUIDE TO THIS BOOK **xxxvii**

***Strategic Modeling***

A Bounded Context is a conceptual boundary where a domain model is applica ble. It provides a context for the Ubiquitous Language that is spoken by the team and expressed in its carefully designed software model, as shown in Figure G.1.

Ubiquitous Language (1)

modeled inside

***Equities domain model with a***

***single, clean Ubiquitous Language***

**Equities Context**

Bounded Context (2)

Explicit boundary around model

**Figure G.1** A diagram illustrating a Bounded Context and relevant

Ubiquitous Language

As you practice strategic design, you’ll find that the **Context Mapping (3)** patterns seen in Figure G.2 work in harmony. Your team will use Context Maps to understand their project terrain.

We’ve just considered the big picture of DDD’s strategic design. Understand ing it is imperative.

Bounded Context (2)

**?**

***Equities Domain Model***

**D**

**Equities Context**

***Accounts Domain Model* Accounts Context**

**?**

**U**

Context Mappings (3) with integration

relationships:

Open Host Service, Published Language,

Anticorruption Layer, Customer-Supplier,

Partnership, Conformist, Shared Kernel

**Figure G.2** Context Maps show the relationships among Bounded Contexts. From the Library of Javier Acero

**xxxviii** GUIDE TO THIS BOOK

***Architecture***

Sometimes a new Bounded Context or existing ones that interact through Con text Mapping will need to take on a new style of **Architecture (4)**. It’s important to keep in mind that your strategically and tactically designed domain models should be architecturally neutral. Still, there will need to be some architecture around and between each model. A powerful architectural style for hosting a Bounded Context is **Hexagonal**, which can be used to facilitate other styles such as **Service-Oriented**, **REST** and **Event-Driven**, and others. Figure G.3 depicts a Hexagonal Architecture, and while it may look a little busy, it’s a fairly simplistic style to employ.

Sometimes we may be tempted to place too much emphasis on architecture rather than focusing on the importance of carefully crafting a DDD-based model. Architecture is important, but architectural influences come and go. Remember to prioritize correctly, placing more emphasis on the domain model, which has greater business value and will be more enduring.

Architecture (4) such as

the Hexagonal style

Tactical domain model at the

**Adapter**

**Adapter**

**Adapter**

**Application**

**Adapter**

**Domain Model**

heart of the Bounded Context

**Adapter**

**Adapter**

**Adapter**

**Adapter**

**Figure G.3** The Hexagonal Architecture with the domain model at the heart of the software

From the Library of Javier Acero

GUIDE TO THIS BOOK **xxxix**

***Tactical Modeling***

*We model tactically inside a Bounded Context* using DDD’s building block patterns. One of the most important patterns of tactical design is **Aggregate (10)**, as illustrated in Figure G.4.

An Aggregate is composed of either a single **Entity (5)** or a cluster of Entities and **Value Objects (6)** that must remain transactionally consistent throughout the Aggregate’s lifetime. Understanding how to effectively model Aggregates is quite important and one of the least well understood techniques among DDD’s building blocks. If they are so important, you may be wondering why Aggre

gates are placed later in the book. First of all, the placement of tactical pat terns in this book follows the same order as is found in [Evans]. Also, since Aggregates are based on other tactical patterns, we cover the basic building blocks—such as Entities and Value Objects—before the more complex Aggre gate pattern.

An instance of an Aggregate is persisted using its **Repository (12)** and later searched for within and retrieved from it. You can see an indication of that in Figure G.4.

Use stateless **Services (7)**, such as seen in Figure G.5, inside the domain model to perform business operations that don’t fit naturally as an operation on an Entity or a Value Object.

Aggregate (10) with transactional

consistency boundary

**Aggregate Type 1**

**Aggregate Type 2**

**<<value object>> Value Type 1**

**<<aggregate root>> Root Entity 1**

**0..\***

**<<value object>> Value Type 2**

**<<aggregate root>> Root Entity 2**

**0..\***

**<<entity>>**

**Entity Type 3**

**<<value object>> Value Type 3**

**<<value object>> Value Type 4**

**<<repository>>**

State inside reflecting true business rules must remain completely consistent

**Repository 1 <<repository>>**

Use a Repository (12) to persist a specific Aggregate type

**Repository 2**

**Figure G.4** Two Aggregate types with their own transactional consistency boundaries From the Library of Javier Acero

**xl** GUIDE TO THIS BOOK

Use a Service (7) to perform an operation that cuts across Aggregates, for example

Query

operation

**<<service>>**

**<<aggregate root>> Root Entity 1**

**Domain Service 1**

Command

operation

**<<aggregate root>> Root Entity 2**

**Figure G.5** Domain Services carry out domain-specific operations, which may involve multiple domain objects.

Use **Domain Events (8)** to indicate the occurrence of significant happenings in the domain. Domain Events can be modeled a few different ways. When they capture occurrences that are a result of some Aggregate command opera tion, the Aggregate itself publishes the Event as depicted in Figure G.6.

Although often given little thought, it’s really important to design **Modules (9)** correctly. In its simplest form, think of a Module as a package in Java or a namespace in C#. Remember that if you design your Modules mechanically rather than according to the Ubiquitous Language, they will probably do more harm than good. Figure G.7 illustrates how Modules should contain a limited set of cohesive domain objects.

Of course, there’s much more to implementing DDD, and I won’t try to cover it all here. There’s a whole book ahead of you that does just that. I think this Guide gets you off on the right foot for your journey through implement ing DDD. So, enjoy the journey!

**Event**

**create**

**Aggregate**

**publish**

**Subscriber**

**handle**

**Event Publisher Event**

**Subscriber**

**Subscriber**

**Figure G.6** Domain Events can be published by Aggregates.

From the Library of Javier Acero

GUIDE TO THIS BOOK **xli com.companyname.context.domain.model.concept**

**<<aggregate root>> Entity 1**

**<<value object>> Identity**

**<<entity>>**

**Entity 2**

**<<value object>> Value Type**

**Figure G.7** A Module contains and organizes cohesive domain objects.

Oh, and just to get you familiarized with Cowboy Logic, here’s one for the trail:

**Cowboy Logic** 

AJ: “Don’t worry about bitin’ off more than you can chew.

Your mouth is probably a whole lot bigger than you

think.” ;-)

LB: “You meant to say ‘mind,’ J. Your mind is bigger than

you think!”

From the Library of Javier Acero

*This page intentionally left blank*

From the Library of Javier Acero

**Chapter 1**

**Getting Started with DDD**

*Design is not just what it looks like and feels like.*

*Design is how it works.*

*—Steve Jobs*

We strive to produce quality in the software we develop. We achieve some qual ity by using tests to help us avoid delivering software with a fatal number of bugs. Yet, even if we could produce completely bug-free software, that in itself does not necessarily mean that a quality software model is designed. The soft ware model—the *way* the software expresses the solution to the business goal being sought—could still suffer greatly. Delivering software with few defects is obviously good. Still, we can reach higher for a well-designed software model that explicitly reflects the intended business objective, and our work may even reach the level of *great*.

The software development approach called *Domain-Driven Design*, or *DDD*, exists to help us more readily succeed at achieving high-quality software model designs. When implemented correctly, DDD helps us reach the point where *our design is exactly how the software works*. This book is about help

ing you correctly implement DDD.

You may be completely new to DDD, you may have tried it and struggled, or you may have already succeeded with it before. Regardless, you no doubt are reading this book because you want to improve your ability to implement DDD, and you can. The chapter road map helps you target your specific needs.

**Road Map to This Chapter**

• Discover what DDD can do for your projects and your teams as you grapple with complexity.

• Find out how to score your project to see if it deserves the DDD investment. • Consider the common alternatives to DDD and why they often lead to problems.

• Grasp the foundations of DDD as you learn how to take the first steps on your project.

• Learn how to sell DDD to your management, domain experts, and technical team members.*continues*

**1**

From the Library of Javier Acero

**2** Chapter 1 GETTING STARTED WITH DDD

• Face the challenges of using DDD armed with knowledge of how to succeed.

• Look in on a team that is learning how to implement DDD.

What should you expect from DDD? Not a heavy, dense, ceremonial process that blocks your way to progress. Rather, expect to use the agile development techniques you probably already have come to trust. Beyond agile, anticipate the acquisition of methods that help you gain deep insight into your business domain, with the prospect of producing testable, malleable, organized, care

fully crafted, high-quality software models.

DDD gives you both the *strategic and tactical modeling tools* necessary to design high-quality software that meets core business objectives.

**Can I DDD?**

You can implement DDD if you have

• A passion for creating excellent software every day, and the tenacity to

achieve that goal

• The eagerness to learn and improve, and the fortitude to admit you need to

• The aptitude to understand software patterns and how to properly apply

them

• The skill and patience to explore design alternatives using proven agile

methods

• The courage to challenge the status quo

• The desire and ability to pay attention to details, to experiment and

discover

• A drive to seek ways to code smarter and better

I’m not going to tell you that there isn’t a learning curve. To put it bluntly, the learning curve can be steep. Yet, this book has been put together to help flatten the curve as much as possible. My goal is to help you and your team implement DDD with the greatest potential for success.

DDD isn’t first and foremost about technology. In its most central principles, DDD is about discussion, listening, understanding, discovery, and business

From the Library of Javier Acero

CAN I DDD? **3**

value, all in an effort to centralize knowledge. If you are capable of *under standing the business* in which your company works, you can at a minimum participate in the software model discovery process to produce a **Ubiquitous Language**. Sure, you’re going to have to learn more about the business, lots more. Still, you are on your way to succeeding with DDD already because you can comprehend the concepts of your business, you revel in developing great

software, and that gives you the proper footing to take DDD all the way. Won’t having years, even a decade or two, of software development expe rience help? It might. Nevertheless, software development experience doesn’t give you the ability to listen and learn from *domain experts*, the people who know the most about some high-priority area of the business. You are at a greater advantage if you can engage with those who seldom, if ever, express themselves using technical lingo. You’re going to have to listen and listen care fully. You’re going to have to respect their viewpoint and trust that they know a lot more than you do.

**There Are Big Advantages to Engaging with Domain Experts**

You are at a greater advantage if you can engage with those who seldom, if ever, express themselves using technical lingo. Just as you are going to learn from them, there is a high probability that they are also going to learn from you.

What you may like best about DDD is that the domain experts are also going to *have to listen to you*. You are on the team just as they are. As strange as it may seem, the domain experts don’t know everything about their business, and they are also going to learn more about it. Just as you are going to learn from them, there is a high probability that they are also going to learn from you. Your questions about what they know will most likely also uncover what they don’t know. You’ll be directly involved in helping everyone on the team discover a deeper understanding of the business, *even shaping the business*.

It’s great when a team learns and grows together. If you give it a chance, DDD makes that possible.

**But We Don’t *Have* Domain Experts**

A domain expert is not one by job title. These are the people who know the line of business you are working in really well. They probably have a lot of background in the business domain, and they might be product designers or even your salespeople.

Look past the job title. The people you are looking for know more about what you are working on than anyone else, and for sure way more than you know. *Find them. Listen. Learn. Design in code.*

So far we’re off to a pretty reassuring start. Still, I am also not going to tell you that technical ability isn’t important, that somehow you can get by without

From the Library of Javier Acero

**4** Chapter 1 GETTING STARTED WITH DDD

it. You will have to grasp some advanced software *domain modeling* concepts. Even so, it doesn’t necessarily mean you are going to be in over your head. If you have abilities somewhere between grasping *Head First Design Patterns* [Freeman et al.] and grokking the original *Design Patterns* [Gamma et al.] text, or even more advanced patterns, you stand a really good chance of succeeding with DDD. You can bank on this: I’m going to do everything I can to make that happen by lowering the bar, no matter what your level of experience.

**What’s a Domain Model?**

It’s a software model of the very specific business domain you are working in. Often it’s implemented as an object model, where those objects have both data and behav

ior with literal and accurate business meaning.

Creating a unique, carefully crafted domain model at the heart of a core, strate

gic application or subsystem is essential to practicing DDD. With DDD your domain models will tend to be smallish, very focused. Using DDD, you never try to model the whole business enterprise with a single, large domain model. Phew, that’s good!

Consider the following perspectives of the people who can benefit from

DDD. I know you fit in here somewhere:

• ***Newbie, junior developer*:** “I’m young, with fresh ideas, I’ve got pent-up

energy to code, and I’m going to have an impact. What’s got me miffed is

one of the projects I sprint on. I didn’t expect that my first gig off campus

would mean shoveling data back and forth using lots of almost identical

yet redundant ‘objects.’ Why is this architecture so complex if that’s all

that’s happening? What’s up with *that*? The code breaks a lot when I try

to change it. Does anyone actually understand what it’s supposed to do?

Now there are some complex new features I have to add. I regularly slap

an *adapter* around legacy classes to shield me from the goo. *No joy*. I’m

sure there’s something I can do besides code and debug all day and night

just to finish iterations. Whatever that is, I’m going to track it down and

own it. I heard some of the others talking about DDD. *It sounds like*

*Gang of Four, but tuned for the domain model.* Nice.”

Gotcha covered.

• ***Midlevel developer*:** “Over the past few months I’ve been included on

the new system. It’s my turn to make a difference. I get it, but what I’m

missing are profound insights when I’m meeting with the senior develop

ers. Sometimes things seem whacked, but I’m not sure why. I’m going to

help change the way things are done around here. I know that throwing

technology at a problem only takes you so far, and that’s basically not far

enough. What I need is *a sound software development technique* that’s

From the Library of Javier Acero

CAN I DDD? **5**

going to help me become a wise and experienced software practitioner. One of the senior architects, the new guy, made a pitch for something called DDD. I’m listening.”

You’re sounding senior already. Read on. Your forward-thinking attitude will be rewarded.

• ***Senior developer, architect*:** “I’ve used DDD on a few projects, but not since landing this new position. I like the power of the *tactical patterns*, but there’s a lot more I could apply, with *strategic design* being one. What I found most insightful when reading [Evans] was the Ubiquitous Lan guage. *That’s powerful stuff*. I’ve had discussions with a number of my teammates and management, trying to influence DDD’s adoption here. One of the new kids and a few of the midlevel and senior members are jazzed about the prospects. Management isn’t so excited. I recently joined this company, and although I was brought in to lead, it seems that the organization is less interested in disruptive advancements than I thought. Whatever. I’m not giving up. With other developers psyched about it, *I know we can make it happen*. The payoffs are going to be much greater than anticipated. We’ll draw the pure business people—the domain experts—closer to our technical teams, and *we’ll actually invest in our solutions*, not just grunt them out iteration after iteration.”

Now *that’s* what a leader does. This book has lots of guidance that shows how to succeed with *strategic design*.

• ***Domain expert*:** “I’ve been involved in specifying the IT solutions to our business challenges for a long time now. Maybe it’s too much to expect, but I wish the developers understood better what we do here. They’re always talking down to us like we’re stupid. What they don’t understand is, if it wasn’t for us there wouldn’t be jobs here for them to mess around with computers. The developers always have some strange way of talking about what our software does. If we talk about A, they say it’s really called B. *It’s like we have to have some sort of dictionary and road map on hand every time we try to communicate what we need*. If we don’t let them have their way by calling B what we know is A, they don’t coop erate. We waste so much time in this mode. *Why can’t the software just work the way the real experts think about the business?*”

You’ve got that right. One of the biggest problems is the false need for translation between business people and techies. This chapter is for you. As you’re going to see, *DDD puts you and developers on level ground*.

From the Library of Javier Acero

**6** Chapter 1 GETTING STARTED WITH DDD

And, surprise! You’ve got some developers already leaning your way. Help

them here.

• ***Manager*:** “We are shipping software. It’s not always with the greatest

result, and changes seem to take longer than they should. The developers

keep talking about some domain something-or-another. I’m not sure we

need to get high centered on yet another technique or methodology, like

it’s some kind of silver bullet. I’ve heard all that a thousand times before.

We try, the fad dies, and we are right back to the same-old same-old. I

keep saying that we need to stay the course and stop dreaming, but the

team keeps hounding me. They’ve worked hard, so I owe them a listen.

*They are smart people and they all deserve a chance to improve things*

before they get torqued and move on. I could allow them some time to

learn and adjust if I can get backing from upper management. I think I

could get that approval if I can convince my boss of the team’s claims of

*achieving critical software investment and a centralization of business*

*knowledge*. Truth is, it will make my job easier if *I can do something to*

*inspire trust and cooperation between my teams and business experts*.

Anyway, that’s what I am hearing I can do.”

Good manager!

Whoever you are, here’s an important heads-up. To succeed with DDD *you are going to have to learn something*, and actually a lot of somethings. That shouldn’t be a big deal, though. You are smart and you have to learn all the time. Yet we all face this challenge:

Personally I’m always ready to learn, although I do not always like being taught.

—Sir Winston Churchill

That’s where this book comes in. I’ve tried to make the teaching as pleas

ant as possible while delivering the vital understanding you need to implement DDD with success.

Your question, though, is: “Why should I do DDD?” That’s fair.

**Why You Should Do DDD**

Actually, I’ve already given you some pretty good reasons why DDD makes so much practical sense. At the risk of breaking the DRY principle (“Don’t repeat yourself”), I reiterate them here and also add to the earlier reasons. Does any one hear an echo?

From the Library of Javier Acero

WHY YOU SHOULD DO DDD **7**

• Put domain experts and developers on a level playing field, which pro duces software that makes perfect sense to the business, not just the cod ers. This doesn’t mean merely tolerating the opposite group. It means becoming one cohesive, tight-knit team.

• That “makes sense to the business” thing means investing in the business by making software that is as close as possible to what the business lead ers and experts would create if they were the coders.

• You can actually teach the business more about itself. No domain expert, no C-level manager, no one, ever knows every single thing about the busi ness. It’s a constant discovery process that becomes more insightful over time. With DDD, everybody learns because everybody contributes to dis covery discussions.

• Centralizing knowledge is key, because with that the business is capable of ensuring that understanding the software is not locked in “tribal knowl edge,” available only to a select few, who are usually only the developers.

• There are zero translations between the domain experts, the software developers, and the software. That doesn’t mean maybe some few transla tions. It means zero translations because your team develops a common, shared language that everyone on the team speaks.

• The design is the code, and the code is the design. The design is how it works. Knowing the best code design comes through quick experimental models using an agile discovery process.

• DDD provides sound software development techniques that address both strategic and tactical design. Strategic design helps us understand what are the most important software investments to make, what existing soft ware assets to leverage in order to get there fastest and safest, and who must be involved. Tactical design helps us craft the single elegant model of a solution using time-tested, proven software building blocks.

Like any good, high-yielding investment, DDD has some up-front cost of time and effort for the team. Considering the typical challenges encountered by every software development effort will reinforce the need to invest in a sound software development approach.

***Delivering Business Value Can Be Elusive***

Developing software that delivers true business value is not the same thing as developing ordinary business software. Software that delivers true business value aligns with the business strategic initiatives and bears solutions with

From the Library of Javier Acero

**8** Chapter 1 GETTING STARTED WITH DDD

clearly identifiable competitive advantage—software that is not about technol ogy, but about the business.

Business knowledge is never centralized. Development teams have to bal

ance and prioritize among the needs and requests of multiple stakeholders and engage with many people having diverse skill sets, all with the goal of uncov ering software functional and nonfunctional requirements. After gathering all that information, how can teams be certain that any given requirement delivers true business value? In fact, what are the business values being sought, and how do you uncover them, prioritize them, and realize them?

One of the worst disconnects of a business software development effort is seen in the gap between domain experts and software developers. Generally speaking, true domain experts are focused on delivering business value. On the other hand, software developers are typically drawn to technology and technical solutions to business problems. It’s not that software developers have wrong motivations; it’s just what tends to grab their attention. Even when soft

ware developers engage with domain experts, the collaboration is largely at a surface level, and the software that gets developed often results in a trans lation/mapping between how the business thinks and operates and how the software developer interprets that. The resulting software generally does not reflect a recognizable realization of the mental model of the domain experts, or perhaps it does so only partially. Over time this disconnect becomes costly. The translation of domain knowledge into software is lost as developers transi tion to other projects or leave the company.

A different, yet related problem is when one or more domain experts do not agree with each other. This tends to happen because each expert has more or less experience in the specific domain being modeled, or they are simply experts in related but different areas. It’s also common for multiple “domain experts” to have no expertise in a given domain, where they are more of a busi

ness analyst, yet they are expected to bring insightful direction to discussions. When this situation goes unchecked, it results in blurred rather than crisp men tal models, which lead to conflicting software models.

Worse still is when the technical approach to software development actually wrongly changes the way the business functions. While a different scenario, it is well known that enterprise resource planning (ERP) software will often change the overall business operations of an organization to fit the way the ERP functions. The total cost of owning the ERP cannot be fully calculated in terms of license and maintenance fees. The reorganization and disruption to the business can be far more costly than either of those two tangible fac

tors. A similar dynamic is at play as your software development teams inter pret what the business needs into what the newly developed software actually does. This can be both costly and disruptive to the business, its customers, and

From the Library of Javier Acero

WHY YOU SHOULD DO DDD **9**

its partners. Furthermore, this technical interpretation is both unnecessary and avoidable with the use of proven software development techniques. The solu tion is a key investment.

***How DDD Helps***

DDD is an approach to developing software that focuses on these three pri mary aspects:

1. DDD brings domain experts and software developers together in order to develop software that reflects the mental model of the business experts. This does not mean that effort is spent on modeling the “real world.” Rather, DDD delivers a model that is the most useful to the business. Sometimes useful and realistic models happen to intersect, but to the degree that they diverge, DDD chooses useful.

With this aspect the efforts of domain experts and software developers are devoted to jointly developing a Ubiquitous Language of the areas of the business that they are focused on modeling. The Ubiquitous Language is developed with full team agreement, is spoken, and is directly captured in the model of the software. It is worth reiterating that the team is com

posed of both domain experts and software developers. It’s never “us and them.” It’s always *us.* This is a key business value that allows business know-how to outlive the relatively short initial development efforts that deliver the first few versions of the software, and the teams that produce it. It’s the point where the cost of developing software is a justifiable busi

ness investment, not just a cost center.

This entire effort unifies domain experts who initially disagree with each other, or who simply lack core knowledge of the domain. Further, it strengthens the close-knit team by spreading deep domain insight among all team members, including software developers. Consider this the hands-on training that every company should invest in its knowledge workers.

2. DDD addresses the strategic initiatives of the business. While this stra tegic design approach naturally includes technical analysis, it is more concerned with the strategic direction of the business. It helps define the best inter-team organizational relationships and provides early-warning systems for recognizing when a given relationship could cause software and even project failure. The technical aspects of strategic design have the goal of cleanly bounding systems and business concerns, which pro tects each *business-level service*. This provides meaningful motivations

From the Library of Javier Acero

**10** Chapter 1 GETTING STARTED WITH DDD

for how an overall *service-oriented architecture* or *business-driven archi*

*tecture* is achieved.

3. DDD meets the real technical demands of the software by using tacti cal design modeling tools to analyze and develop the executable software

deliverables. These tactical design tools allow developers to produce soft

ware that is a correct codification of the domain experts’ mental model,

is highly testable, is less error prone (a provable statement), performs

to service-level agreements (SLAs), is scalable, and allows for distrib

uted computing. DDD best practices generally address a dozen or more

higher-level architectural and lower-level software design concerns, with

a focus on recognizing true business rules and data invariants, and pro

tecting the rules from error situations.

Using this approach to software development, you and your team can succeed in delivering true business value.

***Grappling with the Complexity of Your Domain***

We primarily want to use DDD in the areas that are most important to the business. You don’t invest in what can be easily replaced. *You invest in the nontrivial, the more complex stuff, the most valuable and important stuff that promises to return the greatest dividends.* That’s why we call such a model a **Core Domain (2)**. It is these, and in second priority the *significant* **Supporting Subdomains (2)**, that deserve and get the biggest investment. Rightly, then, we need to grasp what *complex* means.

**Use DDD to Simplify, Not to Complicate**

Use DDD to model a complex domain in the simplest possible way. Never use DDD to make your solution more complex.

What qualifies as complex will differ from business to business. Different

companies have different challenges, different levels of maturity, and different software development capabilities. So rather than determining what is *com plex*, it may be easier to determine what is *nontrivial*. Thus, *your team and management will have to determine if a system you are planning to work on deserves the cost of making a DDD investment.*

**DDD Scorecard:** Use Table 1.1 to determine whether your project qualifies for an investment in DDD. If a row on the scorecard describes your project, place the corresponding number of points in the right-hand column. Tally all the points for your project. If it’s 7 or higher, seriously consider using DDD.

From the Library of Javier Acero

d r

a

c

e

r

o

c

S

D

D

D

e

hT

**1.**

**1**

**e**

**l**

**ba**

**T**

***? r***

***e***

***h***

***g***

***i***

***H***

***r***

***o***

***s***

***t***

***n***

***i***

***o***

***P***

***7***

***f***

***o***

***l***

***a***

***t***

***o***

***T***

***a***

***e***

***r***

***o***

***c***

***S***

***t***

***c***

***e***

***j***

***o***

***r***

***P***

***r***

***u***

***o***

***Y***

***s***

***e***

***o***

***D***

**s**

**t**

**hg**

**u**

**o**

**h**

**T**

**g**

**n**

**i**

**t**

**r**

**o**

**p**

**p**

**u**

**S**

**s**

**t**

**n**

**i**

**o**

**P**

**.**

**.**

**.**

**t**

**c**

**e**

**j**

**o**

**r**

**P**

**r**

**u**

**o**

**Y**

**fI**

i

a

r

b-

o

n

a

e

k

i

l

s

m

e

e

s

s

i

h T

0

s

e

i

f

i

la

u

q

y

l

u

r

t

d

n

a

c

i

r

t

n

e

c

-

a

t

a

d

y

l

e

t

e

lp

m

o

c

s

i

n

o

i

t

a

c

i

l

p

p

a

r

u

o

y

fI

e

lp

m

i

s

e

n

i

m

r

e

t

e

d

o

t

y

s

a

e

a

y

ll

a

c

i

s

a

b

s

i

n

o

i

t

a

r

e

p

o

y

r

e

v

e

e

r

e

h

w

,

n

o

i

t

u

l

o

s

D

U

R

C

e

r

u

p

a

r

o

f

’

n

s

i

t

a

h

t

n

o

i

t

a

c

i

l

p

p

a

y

r

e

v

e

u

o

y

,

e

t

e

le

D

r

o

,

e

t

a

dp

U

,

d

a

e

R

,

e

t

a

e

r

C

o

t

y

r

e

u

q

e

s

a

b

a

t

a

d

e

lp

m

i

s

g

n

i

s

u

f

o

t

r

o

ff

e

d

n

a

e

m

i

t

n

o

e

c

a

f

y

t

t

e

r

p

a

t

u

p

o

t

s

de

e

n

t

s

u

j

m

a

e

t

r

u

o

Y

.

D

D

D

de

e

n

t

’

n

o

d

i

r

t

e

m

r

e

ht

o

h

t

i

w

p

u

e

m

o

c

r

u

o

y

t

s

u

r

t

n

a

c

u

o

y

f

i

,

s

dr

o

w

r

e

ht

o

n

I

.

r

o

t

i

d

e

e

l

b

a

t

e

s

a

b

a

t

a

d

a

x

e

lp

m

o

c

s

i

t

a

h

w

n

e

e

w

t

e

b

e

m

o

s

d

n

a

,

t

i

e

t

a

d

p

u

,

e

l

b

a

t

a

o

t

n

i

y

lt

c

e

r

i

d

a

t

a

d

t

r

e

s

n

i

o

t

s

r

e

s

u

s

’

t

a

h

T

.

e

c

a

fr

e

t

n

i

r

e

s

u

a

de

e

n

n

e

v

e

t

’

n

dl

u

o

w

u

o

y

,

t

i

e

t

e

le

d

s

e

m

i

t

n

e

v

e

dl

u

o

c

u

o

y

f

I

.

t

n

a

v

e

l

e

r

y

ll

a

u

t

p

e

c

n

o

c

s

’

t

i

t

u

b

,

c

i

t

s

i

l

a

e

r

t

o

n

,

n

o

i

t

u

l

o

s

a

e

t

a

e

r

c

o

t

lo

o

t

t

n

e

m

p

o

le

v

e

d

e

s

a

b

a

t

a

d

e

lp

m

i

s

a

e

s

u

.

D

D

D

n

o

y

e

n

o

m

d

n

a

e

m

i

t

s

’

y

n

a

p

m

o

c

r

u

o

y

e

t

s

a

w

t

’

n

o

d

g

n

i

kl

a

t

m

a

I

,

r

a

e

l

c

e

b

o

T

1

s

’

t

i

,

s

n

o

i

t

a

r

e

p

o

s

s

e

n

i

s

u

b

r

e

w

e

f

r

o

0

3

t

s

u

j

s

e

r

i

u

q

e

r

m

e

t

s

y

s

r

u

o

y

fI

3

o

t

5

2

t

o

n

,

s

d

o

ht

e

m

s

s

e

n

a

c

i

l

p

p

a

r

u

o

y

t

a

h

t

n

a

e

m

dl

u

o

w

s

i

h

T

.

e

lp

m

i

s

y

t

t

e

r

p

y

l

ba

b

o

r

p

o

ht

e

m

e

l

p

i

t

lu

m

h

t

i

w

hc

a

e

e

s

a

c

e

s

u

r

o

s

e

i

r

o

t

s

r

e

s

u

l

a

t

o

t

0

3

n

a

h

t

e

r

o

m

o

n

e

v

a

h

dl

u

o

w

n

o

i

t

.

x

e

lp

m

o

c

s

s

e

n

i

s

u

b

la

m

i

n

i

m

y

l

n

o

g

n

i

v

a

h

s

w

o

l

f

e

s

o

h

t

f

o

hc

a

e

h

t

i

w

,

s

w

o

lf

a

c

i

l

p

p

a

n

a

hc

u

s

p

o

le

v

e

d

y

li

s

a

e

d

n

a

y

l

k

c

i

u

q

dl

u

o

c

u

o

y

f

I

.

c

i

g

o

l

e

h

t

le

e

f

t

o

n

d

n

a

s

li

a

r

G

d

n

a

y

v

o

o

r

G

r

o

s

li

a

R

n

o

y

bu

R

g

n

i

s

u

n

o

i

t

,

e

g

n

a

h

c

d

n

a

y

t

i

x

e

lp

m

o

c

r

e

v

o

l

o

r

t

n

o

c

d

n

a

r

e

w

o

p

g

n

i

kc

a

l

f

o

n

i

a

p

.

D

D

D

e

s

u

o

t

de

e

n

t

’

n

s

e

o

d

y

l

ba

b

o

r

p

m

e

t

s

y

s

r

u

o

y

e

t

f

o

y

r

e

V

:

*r*

*o*

*t*

*p*

*m*

*e*

*t*

*a*

*e*

*v*

*a*

*C*

2

s

e

i

r

o

t

s

r

e

s

u

0

4

o

t

0

3

f

o

e

g

n

a

r

e

h

t

n

i

e

r

e

h

w

e

m

o

s

t

a

h

t

y

a

s

s

’

t

e

l

o

S

*e*

*W*

.

hg

u

o

n

e

n

o

o

s

de

z

i

n

g

o

r

u

o

Y

.

y

t

i

x

e

lp

m

o

c

dr

a

w

o

t

g

n

i

p

e

e

r

c

e

b

dl

u

o

c

s

w

o

l

f

e

s

a

c

e

s

u

r

o

*d*

*n*

*u*

*t*

*a*

*do*

*o*

*g*

*y*

*ll*

*a*

*e*

*r*

*,*

*y*

*ll*

*a*

*e*

*r*

.

y

r

o

t

i

r

r

e

t

D

D

D

o

t

n

i

g

n

i

t

t

e

g

e

b

t

h

g

i

m

m

e

t

s

y

s

u

a

c

e

b

t

s

u

J

*.*

*t*

*r*

*o*

*ff*

*e*

*f*

*o*

*le*

*v*

*e*

*l*

i

t

a

c

i

l

p

p

a

s

li

a

r

G

r

o

s

li

a

R

a

l

u

o

c

e

s

o

h

t

n

u

r

g

n

o

l

e

h

t

n

I

ptg9460621

**11**

)

*de*

*u*

*n*

*i*

*t*

*n*

*o*

*C*

(

dr

a

c

e

r

o

c

S

D

D

D

e

hT

**1**

**.**

***? r***

***e***

***h***

***g***

***i***

***H***

***r***

***o***

***s***

***t***

***n***

***i***

***o***

***P***

***7***

***f***

***o***

***l***

***a***

***t***

***o***

***T***

***a***

***e***

***r***

***o***

***c***

***S***

***t***

***c***

***e***

***j***

***o***

***r***

***P***

***r***

***u***

***o***

***Y***

**s**

**t**

**hg**

**u**

**o**

**h**

**T**

**g**

**n**

**i**

**t**

**r**

**o**

**p**

**p**

**u**

**S**

**s**

**t**

**n**

**i**

**o**

**P**

**.**

**.**

**.**

**t**

**c**

**e**

**j**

**o**

**r**

**P**

**r**

t

kl

a

w

o

t

ff

o

s

y

a

p

t

i

e

r

e

H

3

t

i

lli

w

,

w

o

n

x

e

lp

m

o

c

e

b

o

t

g

n

i

o

g

t

o

n

s

i

n

o

i

t

a

c

i

l

p

p

a

e

h

t

f

m

o

d

h

t

i

w

s

o

i

r

a

n

e

c

s

e

g

a

s

u

l

a

e

r

li

t

n

u

e

r

u

s

r

o

f

s

i

h

t

w

o

n

k

t

o

n

y

a

m

u

o

Y

?

y

t

i

x

e

lp

m

o

c

n

i

t

r

e

p

x

e

n

i

a

m

o

d

e

r

A

.

s

d

a

e

l

p

u

S

“

e

h

t

n

i

p

e

t

s

a

s

i

e

r

e

h

t

t

u

b

,

t

i

h

t

i

w

g

n

i

kr

o

w

t

r

a

t

s

r

o

f

g

n

i

k

s

a

y

d

a

e

r

l

a

.

.

.

.

1

e

u

r

t

e

h

t

r

e

v

o

c

n

u

p

l

e

h

y

a

m

t

a

h

t

n

m

u

l

o

c

”

s

t

h

g

u

o

h

T

g

i

t

a

c

i

d

n

i

n

a

y

le

k

i

l

s

’

t

i

,

o

s

fI

.

n

o

o

c

e

b

n

o

o

s

lli

w

r

o

y

d

a

e

r

la

n

o

i

t

a

c

i

l

p

p

a

e

h

t

t

a

h

t

ll

a

t

a

t

n

i

h

y

n

a

s

i

e

r

e

h

t

f

I

.

e

r

e

h

l

u

fe

.

hc

a

o

r

p

p

a

D

U

R

C

a

r

a

p

e

b

o

t

e

m

i

t

do

o

g

a

s

’

e

r

e

h

—

y

t

i

x

e

lp

m

o

c

e

t

a

r

e

do

m

n

e

f

e

h

t

h

t

i

w

d

e

r

o

b

o

s

.

.

.

.

2

e

b

y

l

l

a

u

t

c

a

l

l

i

w

t

i

t

a

h

t

n

o

i

t

a

c

i

d

n

i

t

n

e

i

c

i

f

f

u

s

e

b

y

a

m

t

a

h

t

s

’

t

I

?

m

e

h

t

g

n

i

s

s

u

c

s

i

d

r

a

e

b

.

D

D

D

d

r

a

w

o

t

n

a

e

L

.

x

e

l

p

m

o

c

y

l

e

t

a

r

e

d

o

m

n

a

h

t

a

n

a

m

u

o

y

p

le

h

n

a

c

D

D

D

4

a

r

e

v

o

n

e

t

f

o

e

g

n

a

h

c

o

t

g

n

i

o

g

e

r

a

s

e

r

u

t

a

e

f

s

’

n

o

i

t

a

c

i

l

p

p

i

t

r

e

v

o

le

do

m

r

u

o

y

g

n

i

r

o

t

f

o

s

d

n

i

k

e

h

t

t

a

h

t

e

t

a

p

i

c

i

t

n

a

t

’

n

a

c

u

o

y

d

n

a

,

s

r

a

e

y

f

o

r

e

.

e

lp

m

i

s

e

b

ll

i

w

s

e

o

t

de

e

n

o

t

g

n

i

o

g

e

r

a

u

o

Y

5

s

A

.

w

e

n

s

’

t

i

e

s

u

a

c

e

b

**)**

**2**

**(**

**n**

**i**

**a**

**m**

**o**

**D**

e

h

t

d

n

a

t

s

r

e

dn

u

t

’

n

o

do

m

h

t

i

w

t

n

e

m

i

r

e

p

x

e

d

n

a

.

e

r

o

fe

b

s

i

h

t

e

n

o

d

s

a

h

y

d

o

bo

n

,

w

o

n

k

m

a

e

t

r

u

o

y

d

n

a

u

o

y

o

n

o

d

e

r

o

c

s

o

s

l

a

y

l

n

i

a

t

r

e

c

e

u

d

s

e

v

r

e

s

e

d

t

s

a

e

l

t

a

r

o

,

x

e

lp

m

o

c

s

’

t

i

s

n

a

e

m

y

le

k

i

l

t

s

o

.

D

D

D

e

s

u

o

s

,

a

i

r

e

t

i

r

c

f

o

le

v

e

l

e

h

t

e

n

i

m

r

e

t

e

d

o

t

y

n

i

t

u

r

c

s

l

a

c

i

t

y

la

n

a

h

t

i

w

e

c

.

y

t

i

x

e

ptg9460621

**1**

**e**

**l**

**ba**

**T**

***s***

***e***

***o***

***D***

**u**

**o**

**Y**

**fI**

i

n

e

v

E

w

o

r

g

sr

es

u

n

it

r

o

p

it

a

u

ti

s

r

a

c

e

B

v

e

s

a

h

—

di

o

n

er

om

a

e

hT

b

m

u

n

g

n

a

hc

d

u

o

Y

s

a

r

a

f

m

t

a

hT

n

e

g

i

li

d

lp

m

o

c

**12**

WHY YOU SHOULD DO DDD **13**

This scoring exercise may have led your team to these conclusions:

It’s too bad that we can’t shift gears quickly and easily when we discover we are on the wrong side of complexity, no matter if the wrong side is more or less com plex than we thought.

Sure, but that just means that we need to become much better at determining simplicity versus complexity early on in our project planning. That would save us a lot of time, expense, and trouble.

Once we make a major architectural decision and get several use cases deep in development, we are usually stuck with it. We had better choose wisely.

If any of these observations resonates with your team, you are making good use of critical thought.

***Anemia and Memory Loss***

Anemia can be a serious health ailment with dangerous side effects. When the name **Anemic Domain Model** [Fowler, Anemic] was first coined, *it wasn’t meant to be a complimentary term*, as if to say that a domain model that is weak, without the power of inherent behavioral qualities, could possibly be a good thing. Strangely enough, Anemic Domain Models have popped up left and right in our industry. The trouble is that most developers seem to think this is quite normal and would not even acknowledge that a serious condition exists when employed in their systems. It’s a real problem.

Are you wondering if your model is feeling tired, listless, forgetful, clumsy, needing a good shot in the arm? If you’re suddenly experiencing technical hypochondria, here’s a good way to perform a self-examination. You’ll either put yourself at ease or confirm your worst fears. Use the steps in Table 1.2 to perform your checkup.

**Table 1.2** Determine Your Domain Model Health History

**Yes / No**

Does the software you call a “domain model” have mostly public getters and setters, and no business logic or almost none at all—you know, objects that are mostly attri bute value holders?

Are the software components that frequently use your “domain model” the ones that house most of the business logic of your system, and do those heavily invoke the public getters and setters on the “domain model”? You probably call this particular client layer of the “domain model” a **Service Layer** or **Application Layer (4, 14)**. If instead this describes your user interface, answer “Yes” to this question and write a thousand times on a whiteboard that you’ll never, ever do that again.

**Hint: The correct answers are either “Yes” to both questions or “No” to both questions.** From the Library of Javier Acero

**14** Chapter 1 GETTING STARTED WITH DDD

How did you do?

If you answered “No” to both questions, your domain is doing well.

If you answered “Yes” to both questions, your “domain model” is very, very

ill. It’s anemic. The good news is that you can get help for it by reading on.

If you answered “Yes” to one question and “No” to the other question, you

are either in denial or suffering from delusions or another neurological issue

that could be caused by anemia. What should you do if you have conflicting

answers? Go straight back to the first question and run the self-examination

once again. Take your time, but remember that your answer to both ques

tions must be an emphatic “Yes!”

As [Fowler, Anemic] says, an Anemic Domain Model is a bad thing because

you pay most of the high cost of developing a domain model, but you get little or none of the benefit. For example, because of the object-relational impedance mismatch, developers of such a “domain model” spend a lot of time and effort mapping objects to and from the persistence store. That’s a high price to pay while getting little or no benefit in return. I’ll add that what you have is not a domain model at all, but just a data model projected from a relational model (or other database) into objects. It’s an impostor that may actually be closer to the definition of **Active Record** [Fowler, P of EAA]. You can probably simplify your architecture by not being pretentious and just admit that you are really using a form of **Transaction Script** [Fowler, P of EAA].

**Reasons Why Anemia Happens**

So if an Anemic Domain Model is the sickly outcome of a poorly executed design effort, why do so many use it while thinking that their model is experi encing fine health? Certainly it does reflect a procedural programming mental ity, but I don’t think that’s the primary reason. A good portion of our industry is made up of sample code followers, which isn’t bad as long as the samples are quality ones. Often, however, sample code is purposely focused on demon strating some concept or application programming interface (API) feature in the simplest possible way, without concern for good design principles. Yet oversimplified sample code, which usually demonstrates with a lot of getters and setters, is copied every day without a second thought about design.

There is another, older influence. The ancient history of Microsoft’s Visual

Basic had much to do with where we are today. I’m not saying that Visual Basic was a bad language and integrated development environment (IDE), because it’s always been a highly productive environment and in some ways influenced the industry for the good. Of course, some may have avoided its direct influ

ence altogether, but Visual Basic indirectly caught up with just about every software developer eventually. Just note the timeline shown in Table 1.3.

From the Library of Javier Acero

WHY YOU SHOULD DO DDD **15**

**Table 1.3** The Timeline from Behavior Rich to Infamous Anemia

**1980s 1991 1992–1995 1996 1997 1998–**

Objects make an impact due to Smalltalk and C++

Visual Basic properties and property sheets

Visual tools and IDEs become prolific

Java JDK 1.0 released

JavaBean specification

Explosion of reflection-based tools for Java and .NET plat forms based on properties

What I am talking about is the influence of properties and property sheets, both backed by property getters and setters that were made so popular by the original Visual Basic forms designer. All you had to do was place a few custom control instances on a form, fill out their property sheets, and *voilà!* You had a fully functioning Windows application. It took just a few minutes to do that compared to the few days required to program a similar application directly against the Windows API using C.

So what does all that have to do with Anemic Domain Models? *The Java Bean standard was originally specified to assist in the creation of visual pro gramming tools for Java*. Its motivation was to bring the Microsoft ActiveX capabilities to the Java platform. It offered the hope of creating a market full of third-party custom controls of various kinds, just like Visual Basic’s. Soon almost every framework and library jumped on the JavaBean bandwagon. This included much of the Java SDK/JDK as well as libraries such as the popular Hibernate. Specific to our DDD concerns, *Hibernate was introduced to persist domain models.* The trend continued as the .NET platform reached us.

Interestingly, any domain model that was persisted using Hibernate in the early days had to expose public getters and setters for every persistent sim ple attribute and complex association in every domain object. This meant that even if you wanted to design your POJO (Plain Old Java Object) with a behav ior-rich interface, you had to expose your internals publicly so that Hibernate could persist and reconstitute your domain objects. Sure, you could do things to hide the public JavaBean interface, but by and large most developers didn’t bother or even understand why they should have.

**Should I Be Concerned about Using Object-Relational Mappers with DDD?**

The preceding critique of Hibernate is from a historical perspective. For quite a while now Hibernate has supported the use of hidden getters and setters, and even direct field access. I demonstrate in later chapters how to avoid anemia in your mod els when using Hibernate and other persistence mechanisms. So, don’t sweat it.

From the Library of Javier Acero

**16** Chapter 1 GETTING STARTED WITH DDD

Most, if not all, of the Web frameworks also function solely on the JavaBean

standard. If you want your Java objects to be able to populate your Web pages, the Java objects had better support the JavaBean specification. If you want your HTML forms to populate a Java object when submitted to the server side, your Java form object had better support the JavaBean specification.

Just about every framework on the market today requires, and therefore

promotes, the use of public properties on simple objects. Most developers can’t help but be influenced by all the anemic classes all over their enterprises. Admit it. You’ve been bitten by it, haven’t you? As a result, we have a situation that might be best labeled *anemia everywhere*.

**Look at What Anemia Does to Your Model**

All right, so let’s say we can agree that this is both true and vexing to us. What does *anemia everywhere* have to do with *memory loss*? When you are reading through the client code of an Anemic Domain Model (for example, the impos tor **Application Service (4, 14)**, à la Transaction Script), what do we usually see? Here’s a rudimentary example:

@Transactional

public void saveCustomer(

String customerId,

String customerFirstName, String customerLastName,

String streetAddress1, String streetAddress2,

String city, String stateOrProvince,

String postalCode, String country,

String homePhone, String mobilePhone,

String primaryEmailAddress, String secondaryEmailAddress) {

Customer customer = customerDao.readCustomer(customerId);

if (customer == null) {

customer = new Customer();

customer.setCustomerId(customerId);

}

customer.setCustomerFirstName(customerFirstName);

customer.setCustomerLastName(customerLastName);

customer.setStreetAddress1(streetAddress1);

customer.setStreetAddress2(streetAddress2);

customer.setCity(city);

customer.setStateOrProvince(stateOrProvince);

customer.setPostalCode(postalCode);

customer.setCountry(country);

customer.setHomePhone(homePhone);

customer.setMobilePhone(mobilePhone);

From the Library of Javier Acero

WHY YOU SHOULD DO DDD **17**

customer.setPrimaryEmailAddress(primaryEmailAddress);

customer.setSecondaryEmailAddress (secondaryEmailAddress);

customerDao.saveCustomer(customer);

}

**Example Purposely Kept Simple**

Admittedly, this example is not from a very interesting domain, but it does help us examine a less-than-ideal design and determine how we can refactor it to a much better one. Let’s be clear that this exercise is not leading us to a cooler way to save data. It’s about crafting a software model that adds value to your business, even though this example may not seem valuable.

What did this code just do? Actually it’s pretty versatile code. It saves a Customer no matter whether it is new or preexisting. It saves a Customer no matter whether the last name changed or the person moved to a new home. It saves a Customer no matter whether the person got a new home phone number or discontinued home phone service, or whether he or she got a mobile phone for the first time, or both. It even saves a Customer who switched from using Juno to using Gmail instead, or who changed jobs and now has a new work e-mail address. Wow, this is an awesome method!

Or is it? Actually, we have no idea under what business situations this saveCustomer() method is used—not exactly, anyway. Why was this method created in the first place? Does anyone remember its original intent, and all the motivations for changing it to support a wide variety of business goals? Those memories were quite likely lost only a few weeks or months after the method was created and then modified. And it gets even worse. You don’t believe me? Look at the next version of this same method:

@Transactional

public void saveCustomer(

String customerId,

String customerFirstName, String customerLastName,

String streetAddress1, String streetAddress2,

String city, String stateOrProvince,

String postalCode, String country,

String homePhone, String mobilePhone,

String primaryEmailAddress, String secondaryEmailAddress) {

Customer customer = customerDao.readCustomer(customerId);

if (customer == null) {

customer = new Customer();

customer.setCustomerId(customerId);

}

From the Library of Javier Acero

**18** Chapter 1 GETTING STARTED WITH DDD

if (customerFirstName != null) {

customer.setCustomerFirstName(customerFirstName);

}

if (customerLastName != null) {

customer.setCustomerLastName(customerLastName);

}

if (streetAddress1 != null) {

customer.setStreetAddress1(streetAddress1);

}

if (streetAddress2 != null) {

customer.setStreetAddress2(streetAddress2);

}

if (city != null) {

customer.setCity(city);

}

if (stateOrProvince != null) {

customer.setStateOrProvince(stateOrProvince);

}

if (postalCode != null) {

customer.setPostalCode(postalCode);

}

if (country != null) {

customer.setCountry(country);

}

if (homePhone != null) {

customer.setHomePhone(homePhone);

}

if (mobilePhone != null) {

customer.setMobilePhone(mobilePhone);

}

if (primaryEmailAddress != null) {

customer.setPrimaryEmailAddress(primaryEmailAddress);

}

if (secondaryEmailAddress != null) {

customer.setSecondaryEmailAddress (secondaryEmailAddress);

}

customerDao.saveCustomer(customer);

}

I have to note here that this example isn’t as bad as it gets. Many times the

data-mapping code becomes quite complex, and a lot of business logic gets tucked away in it. I’m sparing you the worst in this example, but you’ve proba bly seen it for yourself.

Now each of the parameters other than the customerId is optional. We

can now use this method to save a Customer under at least a dozen business situations, and more! But is that really a good thing? How could we actually

From the Library of Javier Acero

WHY YOU SHOULD DO DDD **19**

test this method to ensure that it doesn’t save a Customer under the wrong situations?

Without going into extensive detail, this method could function incorrectly in more ways than it could correctly. Perhaps there are database constraints that prevent a completely invalid state from being persisted, but now you have to look at the database to be sure. Almost certainly it will take you some time to mentally map between Java attributes and column names. Once you’ve figured out that part, you find that the database constraints are missing or incomplete.

You could look at the possibly many clients (not counting those added after the user interface was completed to manage automatic remote clients) and com pare source revisions to gain some insight into why it is implemented the way it is right now. As you search for answers, you learn that nobody can explain why this one method works the way it does, or how many correct uses there are. It could take several hours or days to understand it on your own.

**Cowboy Logic** 

AJ: “That fella’s so confused, he doesn’t know if he’s

sackin’ potatoes or rollerskatin’ in a buffalo herd.”

Domain experts can’t help here because they would have to be programmers to understand the code. Even if a domain expert or two knew enough about programming or could at least read the code, they would probably be at least equally at a loss as a developer regarding all that code is meant to support. With all these concerns in mind, do we dare change this code in any way, and if so, how?

There are at least three big problems here:

1. There is little intention revealed by the saveCustomer() interface. 2. The implementation of saveCustomer() itself adds hidden complexity.

3. The Customer “domain object” isn’t really an object at all. It’s really just a dumb data holder.

Let’s call this unenviable situation *anemia-induced memory loss*. It happens all the time on projects that produce this kind of implicit, completely subjective code “design.”

From the Library of Javier Acero

**20** Chapter 1 GETTING STARTED WITH DDD

**Hold On a Minute!**

At this point some of you may be thinking, “Our designs never really leave the

whiteboard. We just draw some structure, and once agreement on that is reached,

we are set free to implement. Scary.”

If so, try not to distinguish design from implementation. Remember that when

practicing DDD, *the design is the code and the code is the design*. In other words,

whiteboard diagrams aren’t the design, just a way to discuss the challenges of the

model.

Stay tuned, as you’ll learn how to take ideas off the whiteboard and make them

work for you.

By now you should be worried about this kind of code and how you can

create a better design. The good news is that you can succeed in producing an explicit, carefully crafted design in your code.

**How to Do DDD**

Let’s back away from heavy implementation discussions for a moment to con sider one of the most empowering features of DDD, the Ubiquitous Language. It’s one of the two primary pillars of DDD’s strengths, the second being the **Bounded Context (2)**, and one cannot properly stand without the other.

**Terms in a Context**

For now think of a Bounded Context as a conceptual boundary around a whole

application or finite system. The reason for this boundary is to highlight that every

use of a given domain term, phrase, or sentence—the Ubiquitous Language—inside

the boundary has a specific contextual meaning. Any use of the term outside that

boundary could, and probably does, mean something different. Chapter 2 explains

Bounded Context in depth.

***Ubiquitous Language***

The Ubiquitous Language is a shared team language. It’s shared by domain experts and developers alike. In fact, it’s shared by everyone on the project team. No matter your role on the team, since you are on the team you use the Ubiquitous Language of the project.

**So, You Think You Know What a Ubiquitous Language Is**

*Obviously it’s the language of the business.*

Well, no.

*Surely it must be adopting industry standard terminology.*

No, not really.

From the Library of Javier Acero

HOW TO DO DDD **21**

*Clearly it’s the lingo used by the domain experts.*

Sorry, but no.

*The Ubiquitous Language is a shared language developed by the team—a team composed of both domain experts and software developers.*

That’s it. Now you’ve got it!

Naturally, the domain experts have a heavy influence on the Language because they know that part of the business best and may be influenced by industry stan dards. However, the Language is *more centered on how the business itself thinks and operates*. Also, many times two or more domain experts disagree on concepts and terms, and they are actually wrong about some because they haven’t thought of every case before. So, as the experts and developers work together to craft a model of the domain, they use discussion with both consensus and compromise to achieve the very *best Language for the project*. The team never compromises on the quality of the Language, just on the best concepts, terms, and meanings. Initial consensus is not the end, however. The Language grows and changes over time as tiny and large breakthroughs are achieved, much like any other living language.

This is no gimmick to get developers to be on the same page as domain experts. It’s not just a bunch of business jargon being forced on developers. It’s a real language that is created by the whole team—domain experts, developers, business analysts, everyone involved in producing the system. The Language may start out with terms that are the natural lingo of the domain experts, but it isn’t limited to that because the Language must grow over time. Suffice it to say that when multiple domain experts are involved in creating the Language, they often disagree ever so slightly on the terms and meanings of what they thought were already ubiquitous.

In Table 1.4, we not only model the administration of flu vaccines in code, but the team must also speak the Language openly. When the team discusses this aspect of the model, they literally speak phrases such as “Nurses adminis ter flu vaccines to patients in standard doses.”

There will be some haggling and wrangling over the Language that exists in the minds of experts and what evolves from there. It’s all part of the nat ural progression of developing the best Language that will matter a lot for a long time. This happens through open discussion, looking at existing docu ments, business tribal knowledge that finally surfaces, as well as referencing standards, dictionaries, and thesauruses. There’s also a point reached where we come to terms with the fact that some words and phrases just don’t aptly fit the business context as well as we once thought, and we realize that others fit it much better.

From the Library of Javier Acero

**22** Chapter 1 GETTING STARTED WITH DDD

**Table 1.4** Analyzing the Best Model for the Business

***Which is better for the business?***

***Though the second and third statements are similar, how should the code be designed?* Possible Viewpoints Resulting Code**

*“Who cares? Just code it up.”* Um, not even close.

*“We give flu shots to patients.”* Better, but misses some

important concepts.

*“Nurses administer flu vaccines to patients in standard doses.”* This seems like what we’d like to run with at this time, at least until we learn more.

patient.setShotType(ShotTypes.TYPE\_FLU); patient.setDose(dose);

patient.setNurse(nurse);

patient.giveFluShot();

Vaccine vaccine = vaccines.standardAdultFluDose(); nurse.administerFluVaccine(patient, vaccine);

So how do you capture this all-important Ubiquitous Language? Here are some ways that work as experimentation leads to advancement:

• Draw pictures of the physical and conceptual domain and label them with names and actions. These drawings are mostly informal but may contain some aspects of formal software modeling. Even if your team does some formal modeling with Unified Modeling Language (UML), you want to avoid any kind of ceremony that will bog down discussions and stifle the creativity of the ultimate Language being sought.

• Create a glossary of terms with simple definitions. List alternative terms, including the ones that show promise and the ones that didn’t work, and why. As you include definitions, you cannot help but develop reusable phrases for the Language because you are forced to write in the Language of the domain.

• If you don’t like the idea of a glossary, still capture some kind of doc umentation that includes the informal drawings of important software concepts. Again, the goal here is to force additional Language terms and phrases to surface.

• Since only one or a few team members may capture the glossary or other written documents, circle back with the rest of the team to review the

From the Library of Javier Acero

HOW TO DO DDD **23**

resulting phrases. You won’t always, if ever, agree on all the captured lin guistics, so be agile and ready to edit heavily.

Those are some ideal first steps to coining a Ubiquitous Language that fits your specific domain. However, this is absolutely not the model that you are developing. It’s only the genesis of the Ubiquitous Language that will very soon be expressed in your system’s source code. We are talking Java, or C#, or Scala, or some other programming language of choice. These drawings and docu

ments also don’t address that the Ubiquitous Language will continue to expand and morph over time. The artifacts that originally led us down an inspiring path to developing a useful Ubiquitous Language that was just right for our specialized domain will very likely be rendered obsolete over time. *That’s why in the end it is team speech and the model in the code that are the most endur*

*ing and the only guaranteed current denotations of the Ubiquitous Language.* Since team speech and the code will be the lasting expression of the Ubiq uitous Language, be prepared to abandon the drawings, glossary, and other documentation that will be difficult to keep up-to-date with the spoken Ubiq uitous Language and source code as they are rapidly enhanced. This is not a requirement of using DDD, but it is pragmatic because it becomes impractical to keep all the documentation in sync with the system.

With this knowledge we can redesign the saveCustomer() example. What if we chose to make Customer reflect each of the possible business goals that it must support?

public interface Customer {

public void changePersonalName(

String firstName, String lastName);

public void postalAddress(PostalAddress postalAddress);

public void relocateTo(PostalAddress changedPostalAddress);

public void changeHomeTelephone(Telephone telephone);

public void disconnectHomeTelephone();

public void changeMobileTelephone(Telephone telephone);

public void disconnectMobileTelephone();

public void primaryEmailAddress(EmailAddress emailAddress);

public void secondaryEmailAddress(EmailAddress emailAddress); }

We can argue that this is not the best model for a Customer, but when implementing DDD, questioning the design is expected. As a team we are free to haggle over what is the best model and settle only after we’ve discovered the Ubiquitous Language that is agreed upon. Still, the preceding interface does explicitly reflect the various business goals that a Customer must support, even if the Language could be improved by refinements again and again.

From the Library of Javier Acero

**24** Chapter 1 GETTING STARTED WITH DDD

It’s important to understand too that the Application Service would also be

refactored to reflect the explicit intentions of the business goals at hand. Each Application Service method would be modified to deal with a single use case flow or user story:

@Transactional

public void changeCustomerPersonalName(

String customerId,

String customerFirstName,

String customerLastName) {

Customer customer = customerRepository.customerOfId(customerId);

if (customer == null) {

throw new IllegalStateException("Customer does not exist.");

}

customer.changePersonalName(customerFirstName, customerLastName);

}

This is different from the original example because in that code a single

method was used to deal with many different use case flows or user stories. In the new example we have limited a single Application Service method to deal with changing the personal name of the Customer, and nothing more. Thus, when using DDD, it is our job to refine Application Services accordingly. This implies that the user interface likewise reflects a narrower user goal, which may have previously been true. Now, however, this specific Application Service method doesn’t require its client to pass ten nulls following the first- and last

name parameters.

Doesn’t this new design put your mind at ease? You can read the code and

easily comprehend it. You can also test it and confirm that it does exactly what it is meant to do, and that it doesn’t do anything that it shouldn’t.

Thus, the Ubiquitous Language is a team pattern used to capture the con

cepts and terms of a specific core business domain in the software model itself. The software model incorporates the nouns, adjectives, verbs, and richer expressions formally formulated and spoken by the close-knit team. Both the software and the tests that verify the model’s adherence to the tenets of the domain capture and adhere to this Language, the same one spoken by the team.

**Ubiquitous, but Not Universal**

Some further clarification about the reach of a Ubiquitous Language is in order. There are a few basic concepts that we need to keep carefully in mind:

From the Library of Javier Acero

THE BUSINESS VALUE OF USING DDD **25**

• *Ubiquitous* means “pervasive,” or “found everywhere,” as *spoken among the team and expressed by the single domain model* that the team develops.

• The use of the word *ubiquitous* is not an attempt to describe some kind of enterprise-wide, company-wide, or worldwide, universal domain language.

• There is one Ubiquitous Language per Bounded Context.

• Bounded Contexts are relatively small, smaller than we might at first imagine. A Bounded Context is large enough only to capture the complete Ubiquitous Language of the isolated business domain, and no larger.

• The Language is ubiquitous only within the team that is working on the project that develops in an isolated Bounded Context.

• On a single project that develops a single Bounded Context, there are always one or more additional isolated Bounded Contexts with which it integrates using **Context Maps (3)**. Each of the multiple Bounded Con texts that integrate has its own Ubiquitous Language, even though some terms of each may overlap.

• If you try to apply a single Ubiquitous Language to an entire enterprise, or worse, universally among many enterprises, you will fail.

When you begin a new project in which you are properly using DDD, iden tify the isolated Bounded Context that is being developed. This places an explicit boundary around your domain model. Discuss, research, conceptual ize, develop, and speak the Ubiquitous Language of the isolated domain model within the explicit Bounded Context. Reject all concepts that are not part of the agreed-upon Ubiquitous Language of your isolated Context.

**The Business Value of Using DDD**

If your experience is anything like mine, you know that software developers can no longer pursue technologies and techniques just because they sound cool or intriguing. We must justify everything that we do. I think that has not always been true, but it is a good thing it is true now. I think the best justifica

tion for using any technology or technique is to provide value to the business. If we can establish real, tangible business value, why would the business ever refuse to use what we recommend?

From the Library of Javier Acero

**26** Chapter 1 GETTING STARTED WITH DDD

The business case is strengthened especially if we can demonstrate that the

business values are higher with our recommended approach than with other options.

**Isn’t Business Value Most Important?**

Sure, and perhaps I should have put this subheading “The Business Value of Using

DDD” earlier in the book. But it’s done, now. This subheading could actually be

“How You Can Sell DDD to Your Boss.” Until you are mostly convinced that there

is a real chance that you can actually implement DDD in your company, this book

is just hypothetical. And I don’t want you to read this book as just a theoretical

exercise. Read it as a concrete reality for your company. Then you can become more

excited about how your company can really benefit. So read on.

Let’s consider the very realistic business value of employing DDD. Be sure to

share this openly with your management, domain experts, and technical team members. The value and benefits are summarized here, then I will elaborate. I start off with the less technical benefits.

1. The organization gains a useful model of its domain.

2. A refined, precise definition and understanding of the business is developed.

3. Domain experts contribute to software design.

4. A better user experience is gained.

5. Clean boundaries are placed around pure models.

6. Enterprise architecture is better organized.

7. Agile, iterative, continuous modeling is used.

8. New tools, both strategic and tactical, are employed.

**1. The Organization Gains a Useful Model of Its Domain**

The emphasis of DDD is to invest our efforts in what matters most to the busi ness. We don’t over-model. We focus on the Core Domain. Other models exist to support the Core Domain and are important, too. Yet the supporting mod els may not be given the priority and effort of the Core Domain.

When our focus is on what distinguishes our business from all others, our

mission is well understood and we have the parameters we need to keep on track. We will deliver exactly what is needed to achieve competitive advantage.

From the Library of Javier Acero

THE BUSINESS VALUE OF USING DDD **27**

**2. A Refined, Precise Definition and Understanding of the Business Is Developed**

The business may actually come to understand itself and its mission better than before. I have heard others state that the Ubiquitous Language developed for the business’s Core Domain has found its way into marketing materials. Cer

tainly it should be incorporated in vision documents and mission statements. As the model is refined over time, the business develops a deep understand ing that can serve as an analysis tool. Details surface out of the minds of your domain experts as you are challenged by one another and shaped by technical team partners. These details can help your business analyze the value of the current and future direction, both strategic and tactical.

**3. Domain Experts Contribute to Software Design**

There is business value when the organization grows a deeper understanding of the core business. Domain experts don’t always agree on concepts and ter minology. Sometimes the differences are fostered by different experiences from outside before joining the organization. Sometimes it happens because of the divergent paths taken by each expert within the same organization. Yet when brought together to a DDD effort, the domain experts gain consensus among themselves. This fortifies the effort and the organization as a whole.

Developers now share a common Language as a unified team along with domain experts. They benefit further from the knowledge transfer from the domain experts they work with. As developers inevitably move on, either to a new Core Domain or out of the organization, training and handoffs are eas

ier. The chances of developing “tribal knowledge,” where only a select few understand the model, are reduced. The experts, remaining developers, and new ones continue to share a common knowledge that is available to anyone in the organization who requires it. This advantage exists because there remains an express goal to adhere to the Language of the domain.

**4. A Better User Experience Is Gained**

Often the end user experience can be tuned to better reflect the model of the domain. Domain-Driven is formally “baked in,” influencing human use of the software.

When software leaves too much to the understanding of its users, users must be trained to make a great number of decisions. In essence the users are only transferring the understanding in their minds into data that they enter into forms. The data is then saved to a data store. If users don’t understand exactly

From the Library of Javier Acero

**28** Chapter 1 GETTING STARTED WITH DDD

what is needed, the results are incorrect. Often this leads to guesswork with related lowered productivity until users can figure out the software.

When the user experience is designed to follow the contours of the under

lying expert model, users are led to correct conclusions. The software actually trains the users, which reduces the training overhead to the business. Quicker to productivity with less training—that’s business value.

We next move into more technically driven benefits to the business.

**5. Clean Boundaries Are Placed around Pure Models**

The technical team is discouraged from doing what might appeal more to their programming and algorithmic interests by aligning expectations with business advantage. Purity in direction allows for focus on the potency of the solution, with efforts directed to where they matter the most. Achieving this is very closely connected to understanding the Bounded Context of the project.

**6. Enterprise Architecture Is Better Organized**

When Bounded Contexts are well understood and carefully partitioned, all teams in the enterprise develop an acute understanding of where and why integrations are necessary. The boundaries are explicit, and the relationships between them are as well. The teams that have models that intersect by usage dependency employ Context Maps to establish formal relationships and ways to integrate. This can actually lead to a very thorough understanding of the entire enterprise architecture.

**7. Agile, Iterative, Continuous Modeling Is Used**

The word *design* can evoke negative thoughts in the minds of business manage ment. However, DDD is not a heavyweight, high-ceremony design and devel opment process. DDD is not about drawing diagrams. It is about carefully refining the mental model of domain experts into a useful model for the busi

ness. It is not about creating a real-world model, as in trying to mimic reality. The team’s efforts follow an agile approach, which is iterative and incremen

tal. Any agile process that the team feels comfortable with can be used success fully in a DDD project. The model that is produced is the working software. It is refined continuously until it is no longer needed by the business.

**8. New Tools, Both Strategic and Tactical, Are Employed**

A Bounded Context gives the team a modeling boundary in which to create a solution to a specific business problem domain. Inside a single Bounded

From the Library of Javier Acero

THE CHALLENGES OF APPLYING DDD **29**

Context is a Ubiquitous Language formulated by the team. It is spoken among the team and in the software model. Disparate teams, sometimes each respon sible for a given Bounded Context, use Context Maps to strategically segregate Bounded Contexts and understand their integrations. Within a single modeling boundary the team may employ any number of useful tactical modeling tools: **Aggregates (10)**, **Entities (5)**, **Value Objects (6)**, **Services (7)**, **Domain Events (8)**, and others.

**The Challenges of Applying DDD**

As you implement DDD, you will encounter challenges. So has everyone else who has succeeded at it. What are the common challenges and how do we jus tify using DDD as we face them? I will discuss the more common ones:

• Allowing for the time and effort required to create a Ubiquitous Language • Involving domain experts at the outset and continuously with the project • Changing the way developers think about solutions in their domain

One of the greatest challenges in using DDD can be the time and effort required to think about the business domain, research concepts and termi nology, and converse with domain experts in order to discover, capture, and enhance the Ubiquitous Language rather than coding in techno-babble. If you want to apply DDD completely, with the greatest value to the business, it’s going to require more thought and effort, and it’s going to take more time. That’s the way it is, period.

It can also be a challenge to solicit the necessary involvement from domain experts. No matter how difficult it is, make sure you do. If you don’t get commitment from at least one real expert, you are not going to uncover deep knowledge of the domain. When you do get the domain experts’ involvement, the onus falls back on the developers. Developers must converse with and listen carefully to the true experts, molding their spoken language into software that reflects their mental model of the domain.

If the domain you are working in is truly distinguishing to your business, domain experts have the edge-knowledge locked up in their heads, and you need to draw it out. I’ve been on projects where the real domain experts are hardly around. Sometimes they travel a lot and it can be weeks between one

hour meetings with them. In a small business it can be the CEO or one of the vice presidents, and they have lots of other things to do that may seem more important.

From the Library of Javier Acero

**30** Chapter 1 GETTING STARTED WITH DDD

**Cowboy Logic** 

AJ: “If you can’t rope the big steer, you’re gonna go

hungry.”

Getting domain expert involvement may require creativity . . .

**How to Involve Domain Experts in Your Project**

Coffee. Use that Ubiquitous Language: 

“Hi, Sally, I got you a tall half-skinny half-one-per

cent extra-hot split-quad-shot latte with whip. Do

you have a few minutes to talk about . . . ?”

Learn to use the Ubiquitous Language of C-Level

management: “. . . profits . . . revenues . . . com

petitive edge . . . market domination.” Seriously.

Hockey tickets.

Most developers have had to *change the way they think* in order to properly

apply DDD. We developers are technical thinkers. Technical solutions come easy for us. It’s not that thinking technically is bad. It’s just that there are times when thinking less technically is better. If it’s been our habit to practice software development only in technical ways for years, perhaps now would be a good time to consider a new way of thinking. Developing the Ubiquitous Language of your domain is the best place to start.

**Cowboy Logic** 

LB: “That fella’s boots are too small. If he don’t find him

self another pair, his toes are gonna hurt.”

AJ: “Yep. If you don’t listen, you’re gonna have to feel.”

There’s another level of thought that is required with DDD that goes beyond

concept naming. When we model a domain through software, we are required From the Library of Javier Acero

THE CHALLENGES OF APPLYING DDD **31**

to give careful thought to which model objects do what. It’s about *designing the behaviors of objects*. Yes, we want the behaviors to be named properly to convey the essence of the Ubiquitous Language. But what an object does by means of a specific behavior must be considered. This is a level of effort that goes beyond creating attributes on a class and exposing getters and setters pub

licly to clients of the model.

Let’s now look at a more interesting domain, one that is more challenging than the rudimentary one previously considered. I purposely repeat my previ ous guidance here to reinforce the ideas.

Again, what happens if we simply provide data accessors to our model? To reemphasize, if we only expose the data accessors for our model objects, the results will look much like a data model. Consider the following two exam ples and decide for yourself which of the two requires more thorough design thought, and which produces the greater benefit to its clients. The requirement is in a Scrum model, where we need to commit a backlog item to a sprint. You probably do this all the time, so it’s most likely a familiar domain.

The first example, as is commonly done today, uses attribute accessors:

public class BacklogItem extends Entity {

private SprintId sprintId;

private BacklogItemStatusType status;

...

public void setSprintId(SprintId sprintId) {

this.sprintId = sprintId;

}

public void setStatus(BacklogItemStatusType status) {

this.status = status;

}

...

}

As for the client of this model:

// client commits the backlog item to a sprint

// by setting its sprintId and status

backlogItem.setSprintId(sprintId);

backlogItem.setStatus(BacklogItemStatusType.COMMITTED);

The second example uses a domain object behavior that expresses the Ubiq uitous Language of the domain:

From the Library of Javier Acero

**32** Chapter 1 GETTING STARTED WITH DDD

public class BacklogItem extends Entity {

private SprintId sprintId;

private BacklogItemStatusType status;

...

public void commitTo(Sprint aSprint) {

if (!this.isScheduledForRelease()) {

throw new IllegalStateException(

"Must be scheduled for release to commit to sprint.");

}

if (this.isCommittedToSprint()) {

if (!aSprint.sprintId().equals(this.sprintId())) {

this.uncommitFromSprint();

}

}

this.elevateStatusWith(BacklogItemStatus.COMMITTED);

this.setSprintId(aSprint.sprintId());

DomainEventPublisher

.instance()

.publish(new BacklogItemCommitted(

this.tenant(),

this.backlogItemId(),

this.sprintId()));

}

...

}

The client of this explicit model seems to operate on safer ground:

// client commits the backlog item to a sprint

// by using a domain-specific behavior

backlogItem.commitTo(sprint);

The first example uses a very data-centric approach. The onus is entirely on

the client to know how to correctly commit the backlog item to a sprint. The model, which is not really a domain model, doesn’t help at all. What if the cli ent mistakenly changes only the sprintId but not the status, or the oppo site? Or what if in the future another attribute must be set? The client code must be analyzed for correct mapping of data values to the proper attributes on the BacklogItem.

This approach also exposes the shape of the BacklogItem object and

clearly focuses attention on its data attributes and not on its behaviors. Even From the Library of Javier Acero

THE CHALLENGES OF APPLYING DDD **33**

if you argue that setSprintId() and setStatus() are behaviors, the case in point is that these “behaviors” have no real business domain value. These “behaviors” do not explicitly indicate the intentions of the scenarios that the domain software is supposed to model, that of committing a backlog item to a sprint. They do cause cognitive overload when the client developer tries to mentally select from among the BacklogItem attributes needed to commit a backlog item to a sprint. There could be many because it’s a data-centric model.

Now consider the second example. Instead of exposing the data attributes to clients, it exposes a behavior that explicitly and clearly indicates that a client may commit a backlog item to a sprint. Experts in this particular domain dis cuss the following requirement of the model:

Allow each backlog item to be committed to a sprint. It may be committed only if it is already scheduled for release. If it is already committed to a different sprint, it must be uncommitted first. When the commit completes, notify inter ested parties.

Thus, the method in the second example captures the Ubiquitous Language of the model in context, that is, the Bounded Context in which the BacklogItem type is isolated. And as we analyze this scenario, we discover that the first solu tion is incomplete and contains bugs.

With the second implementation clients don’t need to know what is required to perform the commit, whether simple or complex. The implementation of this method has as much or as little logic as necessary. We easily added a guard to protect against committing a backlog item that is not yet scheduled for release. True, you can also place guards inside the setters of the first implementation, but the setter now becomes responsible for understanding the full context of the object’s state rather than just the requirements for sprintId and status.

There’s another subtle difference here, too. Note that if the backlog item is already committed to another sprint, it will first be uncommitted from the cur rent sprint. This is an important detail, because when a backlog item is uncom mitted from a sprint, a Domain Event is to be published to clients:

Allow each backlog item to be uncommitted from a sprint. When the backlog item is uncommitted, notify interested parties.

The publication of the uncommitted notification is obtained for free just by using the domain behavior uncommitFrom(). Method commitTo() doesn’t even need to know that it notifies. All it needs to know is that it must uncom mit from any current sprint before committing to a new sprint. Additionally, the commitTo() domain behavior also notifies interested parties with an Event as its final step. Without placing this rich behavior in BacklogItem

From the Library of Javier Acero

**34** Chapter 1 GETTING STARTED WITH DDD

we would have to publish Events from the client. That would certainly leak domain logic from the model. Bad.

Clearly, more thought is needed to create the BacklogItem of the second

example than that of the first. Yet the thought needed is not so much greater, and the benefits are so much higher. The more we learn to design in this way, the easier it becomes. In the end, there is certainly more required thought, more effort, more collaboration and orchestration of team efforts, but not so much that DDD becomes heavy. New thought is well worth the effort.

**~~Whiteboard Time~~**

• Using the specific domain you currently work in, think of the common

terms and actions of the model.

• Write the terms on the board.

• Next, write phrases that should be used by your team when you talk

about the project.

• Discuss them with a real domain expert to see how they could be refined

(remember to bring the coffee).

***Justification for Domain Modeling***

*Tactical modeling* is generally more complex than *strategic modeling*. Thus, if you intend to develop a domain model using the DDD tactical patterns (Aggre gates, Services, Value Objects, Events, and so forth), doing so will require more careful thought and greater investment. Since this is so, how does an organiza tion justify tactical domain modeling? What criteria can be used to qualify a given project for the extra investment needed to properly apply DDD from top to bottom?

Picture yourself leading an expedition through unfamiliar territory. You

would want to understand the surrounding landmasses and borders. Your team would study maps, maybe even draw their own, and determine their stra tegic approach. You would consider aspects of the terrain and how it could be used to your advantage. No matter how much planning is done, some facets of such an endeavor are going to be really difficult.

If your strategy indicated that you’d have to scale a vertical rock face, you’d

need some fitting tactical tools and maneuvers for that ascent. Standing at the bottom and looking up, you might see some indication of specific challenges and perilous areas. Yet, you wouldn’t see every detail until you were on the

From the Library of Javier Acero

THE CHALLENGES OF APPLYING DDD **35**

rock face. You might need to drive pitons into slick rock, but you could use var ious-size cams to wedge into natural cracks. To latch on to these climbing pro tections, you’d bring along your carabiners. You would try to take as straight a path as possible but would have to make specific determinations point by point. Sometimes you might even have to backtrack and reroute depending on what the rock dictated. Many people think of climbing as a dangerous thrill sport, but those who actually climb will tell you it’s safer than driving a car or flying an airplane. Clearly, for that to be true, climbers need to understand the tools and techniques and how to judge the rock.

If developing a given **Subdomain (2)** requires such a difficult, even precari ous, ascent, we’d bring the DDD tactical patterns along for the climb. A busi ness initiative that matches the criteria of the Core Domain should not quickly dismiss the use of the tactical patterns. The Core Domain is an unknown and complex area. The team is best protected against a disastrous mid-asset fall if using the right tactics.

Here’s some practical guidance. I begin with the high-level ones and prog ress to more details:

• If a Bounded Context is being developed as the Core Domain, it is stra tegically vital to the success of the business. The core model is not well understood and will require lots of experimentation and refactoring. It likely deserves commitment to longevity with continuous enhancement. It may not always be your Core Domain. Nonetheless, if the Bounded Context is complex, innovative, and needs to endure for a long time as it undergoes change, strongly consider the use of the tactical patterns as an investment in the future of your business. This assumes that your Core Domain deserves the best developer resources with a high skill level.

• A domain that may become a **Generic Subdomain (2)** or Supporting Sub domain to its consumers may actually be a Core Domain to your busi ness. You don’t always judge a domain from the viewpoint of its ultimate consumers. If you are developing a Bounded Context as your chief busi ness initiative, it is your Core Domain regardless of how it is viewed by customers outside your business. Strongly consider the use of the tactical patterns.

• If you are developing a Supporting Subdomain that, for various reasons, cannot be acquired as a third-party Generic Subdomain, it is possible that the tactical patterns would benefit your efforts. In this case consider the skill level of the team and whether or not the model is new and inno

vative. It is innovative if it adds specific business value, captures special knowledge, and is not just technically intriguing. If the team is capable of

From the Library of Javier Acero

**36** Chapter 1 GETTING STARTED WITH DDD

properly applying tactical design, and the Supporting Subdomain is inno

vative and must endure for years in the future, this is a good opportunity

to invest in your software using tactical design. However, this does not

make this model the Core Domain since in the eyes of the business it is

merely Supporting.

These guidelines may be somewhat confining if your business employs a

good number of developers with vast experience in and a very high comfort level with domain modeling. Where experience is very high, and the engineers themselves believe the tactical patterns would be the best choice, it makes sense to trust their opinion. Honest developers, no matter how experienced, will indicate in a specific case that developing a domain model is, or is not, the best choice.

The type of business domain itself is not automatically the determining fac

tor for choosing a development approach. Your team should consider import ant questions to help you make the final determination. Consider the following short list of more detailed decision parameters, which is more or less aligned with and expands on the preceding higher-level guidelines:

• Are domain experts available and are you committed to forming a team

around them?

• Although the specific business domain is somewhat simple now, will it

grow in complexity over time? There is risk in using Transaction Script1

for complex applications. If you use Transaction Script now, will the

potential for refactoring to a behavioral domain model later on be practi

cal if/when the Context becomes complex?

• Will the use of the DDD tactical patterns make it easier and more prac

tical to integrate with other Bounded Contexts, whether third-party or

custom developed?

• Will development really be simpler and require less code if you use Trans

action Script? (Experience with both approaches proves that many times

Transaction Script requires as much or more code. This is probably

because the complexity of the domain and the innovation of the model

were not well understood during project planning. Underestimating

domain complexity and the innovation involved happens often.)

• Do the critical path and timeline allow for any overhead required for tac

tical investment?

1. Here I am generalizing terms. In this list I use Transaction Script to represent sev eral non-domain-model approaches.

From the Library of Javier Acero

THE CHALLENGES OF APPLYING DDD **37**

• Will the tactical investment in a Core Domain protect the system from changing architectural influences? Transaction Script may leave it exposed. (Domain models are often enduring while architectural influences tend to be more disruptive to other layers.)

• Will clients/customers benefit from a cleaner, enduring design and devel opment approach, or could their application be replaced by an off-the shelf solution tomorrow? In other words, why would we ever develop this as a custom application/service in the first place?

• Will developing an application/service using tactical DDD be more diffi cult than using other approaches such as Transaction Script? (Skill level and availability of domain experts is vital to answering this question.)

• If the team’s toolkit was complete with DDD enablers, would we consci entiously choose to use another approach instead? (Some enablers make model persistence practical, such as using object-relational mapping, full Aggregate serialization and persistence, an Event Store, or a framework that supports tactical DDD. There may be other enablers, too.)

This list is not prioritized for your domain, and you can probably assemble additional criteria. You understand the compelling reasons for using the best and most empowering methods possible to your advantage. You also know your business and technology landscape. In the end it is the business customer, not the object practitioners and technologists, who must be pleased. Choose wisely.

***DDD Is Not Heavy***

In no way do I want to imply that properly practicing DDD leads to a heavy weight process with lots of ceremony and all the crufty documentation arti facts that must be supported. That’s not what DDD is about. It is meant to fit well into any agile project framework, such as Scrum, that the team desires to use. Its design tenets lean toward rather rapid test-first refinements of a real software model. If you were in need of developing a new domain object, such as an Entity or a Value Object, the test-first approach works like this:

1. Write a test that demonstrates how the new domain object should be used by a client of the domain model.

2. Create the new domain object with enough code to make the test compile.

3. Refactor both until the test properly represents the way a client would use the domain object, and the domain object has proper behavioral method signatures.

From the Library of Javier Acero