

contents

foreword xiii
preface xv
acknowledgments xvii
about this book xix
about the author xxiii

I *Functional domain modeling: an introduction* 1

- 1.1 What is a domain model? 3
- 1.2 Introducing domain-driven design 4
 - The bounded context* 5 ▪ *The domain model elements* 5
 - Lifecycle of a domain object* 9 ▪ *The ubiquitous language* 14
- 1.3 Thinking functionally 15
 - Ah, the joys of purity* 18 ▪ *Pure functions compose* 22
- 1.4 Managing side effects 27
- 1.5 Virtues of pure model elements 29
- 1.6 Reactive domain models 32
 - The 3+1 view of the reactive model* 33 ▪ *Debunking the “My model can’t fail” myth* 33
 - Being elastic and message driven* 35
- 1.7 Event-driven programming 36
 - Events and commands* 38 ▪ *Domain events* 39

1.8 Functional meets reactive 41

1.9 Summary 42

2 *Scala for functional domain models* 44

2.1 Why Scala? 45

2.2 Static types and rich domain models 47

2.3 Pure functions for domain behavior 49

Purity of abstractions, revisited 53 ▪ *Other benefits of being referentially transparent* 55

2.4 Algebraic data types and immutability 56

Basics: sum type and product type 56 ▪ *ADTs structure data in the model* 58 ▪ *ADTs and pattern matching* 59
ADTs encourage immutability 60

2.5 Functional in the small, OO in the large 61

Modules in Scala 62

2.6 Making models reactive with Scala 67

Managing effects 67 ▪ *Managing failures* 68
Managing latency 70

2.7 Summary 71

3 *Designing functional domain models* 73

3.1 The algebra of API design 74

Why an algebraic approach? 75

3.2 Defining an algebra for a domain service 76

Abstracting over evaluation 76 ▪ *Composing abstractions* 77
The final algebra of types 79 ▪ *Laws of the algebra* 81
The interpreter for the algebra 82

3.3 Patterns in the lifecycle of a domain model 83

Factories—where objects come from 85 ▪ *The smart constructor idiom* 86 ▪ *Get smarter with more expressive types* 88
Aggregates with algebraic data types 89 ▪ *Updating aggregates functionally with lenses* 92 ▪ *Repositories and the timeless art of decoupling* 97 ▪ *Using lifecycle patterns effectively—the major takeaways* 104

3.4 Summary 105

- 4 Functional patterns for domain models 107**
- 4.1 Patterns—the confluence of algebra, functions, and types 109
 - Mining patterns in a domain model 110* ▪ *Using functional patterns to make domain models parametric 111*
 - 4.2 Basic patterns of computation in typed functional programming 116
 - Functors—the pattern to build on 117* ▪ *The Applicative Functor pattern 118* ▪ *Monadic effects—a variant on the applicative pattern 125*
 - 4.3 How patterns shape your domain model 134
 - 4.4 Evolution of an API with algebra, types, and patterns 139
 - The algebra—first draft 140* ▪ *Refining the algebra 141*
 - Final composition—follow the types 143*
 - 4.5 Tighten up domain invariants with patterns and types 144
 - A model for loan processing 144* ▪ *Making illegal states unrepresentable 146*
 - 4.6 Summary 147
- 5 Modularization of domain models 149**
- 5.1 Modularizing your domain model 150
 - 5.2 Modular domain models—a case study 152
 - Anatomy of a module 152* ▪ *Composition of modules 159*
 - Physical organization of modules 160* ▪ *Modularity encourages compositionality 162* ▪ *Modularity in domain models—the major takeaways 163*
 - 5.3 Type class pattern—modularizing polymorphic behaviors 163
 - 5.4 Aggregate modules at bounded context 166
 - Modules and bounded context 167* ▪ *Communication between bounded contexts 168*
 - 5.5 Another pattern for modularization—free monads 169
 - The account repository 169* ▪ *Making it free 170*
 - Account repository—monads for free 172* ▪ *Interpreters for free monads 175* ▪ *Free monads—the takeaways 178*
 - 5.6 Summary 179

6 *Being reactive* 180

- 6.1 Reactive domain models 181
- 6.2 Nonblocking API design with futures 184
 - Asynchrony as a stackable effect* 185
 - *Monad transformer-based implementation* 187
 - *Reducing latency with parallel fetch—a reactive pattern* 189
 - *Using `scalaz.concurrent.Task` as the reactive construct* 193
- 6.3 Explicit asynchronous messaging 196
- 6.4 The stream model 197
 - A sample use case* 198
 - *A graph as a domain pipeline* 202
 - Back-pressure handling* 204
- 6.5 The actor model 205
 - Domain models and actors* 206
- 6.6 Summary 211

7 *Modeling with reactive streams* 213

- 7.1 The reactive streams model 214
- 7.2 When to use the stream model 215
- 7.3 The domain use case 216
- 7.4 Stream-based domain interaction 217
- 7.5 Implementation: front office 218
- 7.6 Implementation: back office 220
- 7.7 Major takeaways from the stream model 223
- 7.8 Making models resilient 224
 - Supervision with Akka Streams* 225
 - *Clustering for redundancy* 226
 - *Persistence of data* 226
- 7.9 Stream-based domain models and the reactive principles 228
- 7.10 Summary 229

8 *Reactive persistence and event sourcing* 230

- 8.1 Persistence of domain models 231
- 8.2 Separation of concerns 233
 - The read and write models of persistence* 234 ▪ *Command Query Responsibility Segregation* 235
- 8.3 Event sourcing (events as the ground truth) 237
 - Commands and events in an event-sourced domain model* 238
 - Implementing CQRS and event sourcing* 240
- 8.4 Implementing an event-sourced domain model (functionally) 242
 - Events as first-class entities* 243 ▪ *Commands as free monads over events* 245 ▪ *Interpreters—hideouts for all the interesting stuff* 247 ▪ *Projections—the read side model* 252 ▪ *The event store* 253 ▪ *Distributed CQRS—a short note* 253 ▪ *Summary of the implementation* 254
- 8.5 Other models of persistence 255
 - Mapping aggregates as ADTs to the relational tables* 255
 - Manipulating data (functionally)* 257 ▪ *Reactive fetch that pipelines to Akka Streams* 258
- 8.6 Summary 259

9 *Testing your domain model* 260

- 9.1 Testing your domain model 260
- 9.2 Designing testable domain models 262
 - Decoupling side effects* 263 ▪ *Providing custom interpreters for domain algebra* 264 ▪ *Implementing parametricity and testing* 265
- 9.3 xUnit-based testing 266
- 9.4 Revisiting the algebra of your model 267
- 9.5 Property-based testing 268
 - Modeling properties* 268 ▪ *Verifying properties from our domain model* 270 ▪ *Data generators* 274 ▪ *Better than xUnit-based testing?* 277
- 9.6 Summary 278

10	Summary—core thoughts and principles	279
10.1	Looking back	279
10.2	Rehashing core principles for functional domain modeling	280
	<i>Think in expressions</i>	280
	▪ <i>Abstract early, evaluate late</i>	281
	<i>Use the least powerful abstraction that fits</i>	281
	▪ <i>Publish what to do, hide how to do within combinators</i>	282
	▪ <i>Decouple algebra from the implementation</i>	282
	▪ <i>Isolate bounded contexts</i>	283
	<i>Prefer futures to actors</i>	283
10.3	Looking forward	283
	<i>index</i>	285