

FUNCTIONAL AND REACTIVE DOMAIN MODELING

DEBASISH GHOSH

FOREWORD BY JONAS BONÉR

Functional and Reactive Domain Modeling

*Functional and
Reactive Domain
Modeling*

DEBASISH GHOSH



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2017 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Jennifer Stout
Review editor: Aleksandar Dragosavljević
Technical development editor: Alain Couniot
Copyeditor: Sharon Wilkey
Proofreader: Alyson Brener
Technical proofreaders: Thomas Lockney, Charles Feduke
Typesetter: Dennis Dalinnik
Cover designer: Leslie Haimes

ISBN: 9781617292248

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 21 20 19 18 17 16

brief contents

- 1 ■ Functional domain modeling: an introduction 1
- 2 ■ Scala for functional domain models 44
- 3 ■ Designing functional domain models 73
- 4 ■ Functional patterns for domain models 107
- 5 ■ Modularization of domain models 149
- 6 ■ Being reactive 180
- 7 ■ Modeling with reactive streams 213
- 8 ■ Reactive persistence and event sourcing 230
- 9 ■ Testing your domain model 260
- 10 ■ Summary—core thoughts and principles 279

contents

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

1 *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 |

foreword

We developers are drowning in complexity. We need to support a rapidly growing number of highly demanding users producing more and more data, with lower latency and higher throughput, taking advantage of multicore processors and distributed infrastructures. And we have to ship in time under tight deadlines for those ever-demanding customers.

Our jobs have never been easy. In order to stay productive and enjoy our work, we need the right set of tools—tools that can manage the growing complexity and requirements with the optimal use of resources. As always, the answer is not as simple as chasing the newest, shiniest things—even though that’s tempting. We must also look back, learn from the hard-won wisdom of the past, and see if there is a way to apply it to the contexts and challenges of today. I consider domain-driven design (DDD), functional programming (FP), and reactive principles among the most useful tools we’ve developed. Each one can help us to manage an axis of complexity:

- *Domain complexity*—Domain-driven design helps us to mine and understand the distinct characteristics and semantics of the domain. By communicating with stakeholders in their own language, DDD makes it easier to create extensible domain models that map to the real world, while allowing for continuous change.
- *Solution complexity*—Functional programming helps us with reasonability and composability. Through reusable pure functions working on stable (immutable) values, FP gives us a great toolset to reason about time, concurrency, and abstraction through (referentially transparent) code that does not “lie.”

- *System complexity*—Reactive principles, as defined in The Reactive Manifesto (www.reactivemanifesto.org), can help us to manage the increasingly complex world of multicore processors, cloud computing, mobile devices, and the Internet of Things, in which essentially all new systems are distributed systems from day one. It's a vastly different and more challenging world to operate in, but also one with lots of new and interesting opportunities. The shift has forced our industry to rethink some of its old best practices around system architecture and design.

I thoroughly enjoyed reading this book, and it very much represents my own journey over the last 10 years. I started out as an OO practitioner—hacking C++ and Java during the day and reading the classic “Gang of Four” book at night.¹ Then, in 2006, I discovered Eric Evans’s book on domain-driven design,² and it was more or less a revelation. I turned into some kind of DDD aficionado, applying it everywhere I could. A couple years later I started tinkering with Erlang, and later Scala, which made me rediscover and fall in love with functional programming. I had studied FP at university back in the day, but had not understood the true power of it until now. Around this time I had also started to lose faith in enterprise Java’s “best practices” around concurrency, resilience, and scalability. Frustrated and guided by a better way of doing things—the Erlang way, and specifically the actor model³—I started the Akka project, which I believe has helped take reactive principles into the mainstream.

What captured me about this book is that it sets out on the rather bold mission of bringing together these three very different tools—domain-driven design, functional programming, and reactive principles—in a practical way. It teaches you how things like bounded contexts, domain events, functions, monads, applicatives, futures, actors, streaming, and CQRS can help keep complexity under control. This is not a book for the faint of heart. It is demanding. But if you put in the hours, you will be rewarded in spades. Fortunately for you, dear reader, you’ve already taken the first step. All you have to do now is keep on reading.

JONAS BONÉR
FOUNDER AND CTO OF LIGHTBEND
CREATOR OF AKKA

¹ *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994), by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, also known as the “Gang of Four.”

² *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003), by Eric Evans.

³ https://en.wikipedia.org/wiki/Actor_model.

preface

It was the summer of 2014 when Manning Publications expressed interest in an updated version of *DSLs in Action* (<https://www.manning.com/books/dsls-in-action>), because of all the new developments going around in design and implementation of programming languages. Incidentally, around the same time, I was going through a beautiful experience of rearchitecting a complex domain model by using functional paradigms.

With a team of software engineers who had just graduated into the world of functional programming using Scala, I was modeling domain behaviors as pure functions, designing domain objects as algebraic data types, and had started appreciating the values of algebraic API design. Every member of our team had the red bible of *Functional Programming in Scala* (<https://www.manning.com/books/functional-programming-in-scala>) that Paul Chiusano and Rúnar Bjarnason had just written.

Our domain model was complex, and our implementation followed the principles of domain-driven design (DDD) that Eric Evans had described in his esteemed book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003). But instead of using the object-oriented approach, we decided to adopt the functional paradigm. It all started as an experiment, which at the end of the day proved to be quite a successful and satisfying experience. Now when I look back, I find the core tenets of DDD in complete harmony with the generic principles of software engineering. Hence it's no wonder that functional, domain-driven design may emerge as one of the most dominant paradigms of domain modeling.

This book is an appreciation of and testimony to the success that we've had in architecting domain models by using functional programming. I decided to share

the practices that we followed, the principles that we adopted, and the idioms of Scala that we used in our implementation. And Manning readily accepted this proposal and decided to go ahead with the project.

Regardless of how functional your domain model is, one key criterion that defines the success of implementation is the responsiveness of your overall application. No user likes to stare at a waiting cursor on the screen, which, as we've seen from our experience, often results from architectures that unnecessarily block the main thread of execution. Operations that are expensive and could take time to complete need to be executed asynchronously, keeping the main thread free for other user actions. The Reactive Manifesto (www.reactivemanifesto.org) defines characteristics that your model should have in order to ensure that your application is nonblocking and responsive and doesn't suffer from the tyrannies of unbounded latency. This is the other aspect that I decided to address in this book. After lots of discussions with the friendly Manning team, we decided that the book needed to talk about a successful marriage of the functional and reactive paradigms.

And thus was born *Functional and Reactive Domain Modeling*. I had immense fun working on the project and I hope you'll have a similar experience as a reader. I have received innumerable comments from readers, reviewers, and well-wishers that have gone a long way in improving the quality of the book. I enjoyed tremendous support from Manning and its exceptional team of editors and reviewers.

acknowledgments

I would like to thank many people who have directly or indirectly inspired the creation of this book.

First, I would like to thank Martin Odersky, the creator of the Scala programming language that I have used to implement all the paradigms of functional and reactive domain modeling. A big thank you also goes to the creators of Scalaz, the amazing library that has brought the real joy of pure functional programming to the core Scala language.

Twitter has been an amazing community, fostering discussions of varied kinds. I've had a lot of stimulating discussions on functional programming with some of the great minds there. Thanks to each one of those great minds who planted the seeds of thought for writing a book on this topic.

Thanks to all the reviewers: Barry Alexander, Cosimo Attanasi, Daniel Garcia, Jan Nonnen, Jason Goodwin, Jaume Valls, Jean-François Morin, John G. Schwitz, Ken Fricklas, Lukasz Kupka, Michael Hamrah, Othman Doghri, Rintcius Blok, Robert Miller, Saleem Shafi, Tarek Nabil, and William E. Wheeler. Time is possibly the most valuable resource we have, and I feel grateful to them for giving theirs. Each reviewer came up with great suggestions that have helped improve the quality of this book.

Thanks to all the readers who bought the MEAP version, interacted regularly on the Author Online forum, and helped keep me motivated to complete the book. And special thanks to Arya Irani, who contributed a pull request that helped me upgrade the free monad code base from Scalaz 7.1 to 7.2. Also special thanks to

Thomas Lockney and Charles Feduke for doing a thorough technical review of the various MEAP versions.

I also want to thank Manning Publications for repeating its trust in me. I had a great time working with Manning on my first book, and this repeat experience has been even more fun. I want to thank the following staff at Manning for their excellent work:

- Michael Stephens and Christina Rudloff for inspiring me to get started with the project
- Jennifer Stout for her unrelenting perseverance in correcting all my mistakes through the entire arduous journey of the 10 chapters
- Alain Couniot for his insightful technical reviews throughout the journey
- Candace Gillhoolley and Ana Romac, who helped promote this book
- Mary Piergies, Kevin Sullivan, Maureen Spencer, and all the others who worked behind the scenes to turn the rough draft into a real book, including Sharon Wilkey, Alyson Brener, April Milne, and Dennis Dalinnik

Thanks to Jonas Bonér for contributing the foreword to my book. I have been privileged to know Jonas for a long time now, and he has been a major source of inspiration in most of my software development tasks.

And last but not least, I am indebted to my wife, Mou, and my little son, Aarush, for providing me the most fulfilling of ecosystems in which the creative task of writing a book on functional programming was made possible.

about this book

This book is about implementing domain models by using the paradigms of functional programming—and making the models responsive by using reactive principles like nonblocking computations and asynchronous messaging.

A domain model is about the problem domain, and there are many ways you can implement a solution architecture that delivers the same functionalities as the problem domain model. Traditionally, we've seen object-oriented techniques being used while designing domain models. In this book, I use an orthogonal approach—modeling domain behaviors with pure functions and domain entities with algebraic data types, and using immutability as one of the core concerns of the design space. As a reader, you'll learn about functional design patterns based on algebraic techniques that you'll be able to reuse directly while implementing domain models of your own.

The book is also about reactive programming—using futures, promises, actors, and streams to ensure that your model is responsive enough and operates on a budget of bounded latency.

I use the Scala programming language for implementing the domain models in the book. Being a resident of the JVM with strong support of object-oriented and functional programming principles, Scala is one of the most widely used languages today. Even so, the core principles that the book discusses are equally applicable for other functional languages like Haskell.

Roadmap

Chapter 1 starts with an overall discussion of what you'll learn by reading the book. It provides an overview of what I mean by a *domain model* and discusses some of the concepts behind domain-driven design. It talks about the core tenets of functional programming (FP) and the benefits you get by designing your domain models to be referentially transparent and with a clear decoupling of side effects from pure logic. It defines what I mean by a *reactive model* and how you can combine the two principles of FP and reactive design to make your model more responsive and scalable.

Chapter 2 discusses the benefits of using Scala as the implementation language for functional and reactive domain modeling. It talks about the benefits of static typing and how the advanced type system of Scala makes your model more robust and verifiable. In this chapter, you'll also learn how to combine the OO and FP power of Scala to achieve modular and pure models.

Chapter 3 starts with a detailed discussion of algebraic API design. Without committing to an implementation, you can design APIs based on the algebra of the abstractions. This chapter covers the benefits of this approach in gory detail and with quite a few examples from the real-world modeling of personal banking systems. Algebra comes with its laws, and when you build APIs based on algebra, you need to ensure that the laws are honored by your implementation. The chapter concludes with a discussion of some of the lifecycle patterns of domain objects, starting from the time that they come into existence through factories, then perform domain behaviors as aggregates, and finally get persisted into repositories.

Chapter 4 focuses on functional design patterns, quite different from the object-oriented design patterns that you've learned to this point. A functional design pattern is based on an algebra that can have multiple implementations (or interpretations) and hence is far more reusable than an OO design pattern. I discuss functors, applicatives, and monads as the primary reusable patterns from functional programming languages. The chapter also discusses a few use cases for evolving your domain model based on the algebra of these patterns.

Chapter 5 is about modularizing your domain models. One nontrivial domain model is a collection of smaller models, each of them known as a *bounded context*. This chapter explains how to design bounded contexts as separate artifacts and how to ensure that communications across multiple bounded contexts are decoupled in space and time. This is one of the core concepts of domain-driven design and can be realized easily using an asynchronous messaging backbone. This chapter also introduces free monads, another advanced technique of modularization using the principles of functional programming.

Chapter 6 discusses reactive domain models. You'll learn how to design reactive APIs that make models responsive by not blocking the main thread of execution. The chapter presents various forms of nonblocking communication across domain objects and bounded contexts like futures, promises, actors, and reactive streams. The chapter

also discusses a use case for using reactive streams in an example from the domain of personal banking.

Chapter 7 is about reactive streams. I implement a moderately sized use case to demonstrate the power of reactive streams using Akka Streams. While chapter 6 touches on the drawbacks of the actor model, chapter 7 shows how to improve on those drawbacks by implementing typed APIs with Akka Streams.

Chapter 8 covers domain model persistence. The chapter starts with a critique of the CRUD-based persistence model and introduces the notion of reactive persistence using event-driven techniques. I talk about the complete history of domain events that folds into the current state of the model and discuss implementation techniques such as CQRS and event sourcing that lead to a more scalable persistence model. The chapter also demonstrates a CRUD-based implementation using Slick, a popular functional-to-relational mapping framework for RDBMS.

Chapter 9 is about testing domain models. It starts with the classic xUnit-based testing methodologies, and identifies the drawbacks and the scenarios where they can be improved on by using algebraic testing. It introduces property-based testing that allows users to write algebraic properties that will be verified through automatic generation of data during runtime. The chapter discusses implementations of this technique with existing domain models from earlier chapters using ScalaCheck, the property-based testing library for Scala.

The book concludes with a review of core principles and a discussion of future trends in domain modeling in chapter 10.

Code conventions and downloads

All source code in listings or in text is in a fixed-width font like this to separate it from ordinary text. Sometimes we needed to break a line into two or more to fit on the page. The continued line is indicated by this arrow: ➔

Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

The code for the examples in this book is available for download from the publisher's website at <https://www.manning.com/books/functional-and-reactive-domain-modeling> and from GitHub at <https://github.com/debasishg/frdomain>.

Quizzes and exercises

The book comes with a collection of quizzes and exercises that will help readers track their understanding of the materials discussed. Chapters 1 and 2 contain a number of quizzes on the basic concepts. Each numbered “Quiz Time” is followed within a page or two by the answer inline, in the form of a “Quiz Master’s Response.” These elements appear as shown in the examples below:



QUIZ TIME 1.1 What do you think is the primary drawback of this model?



QUIZ MASTER'S RESPONSE 1.1 It's the *mutability* that hits you in two ways: It makes it hard to use the abstraction in a concurrent setting and makes it difficult to reason about your code.

Things become a bit more serious from chapter 3 onward. The quizzes are replaced by numbered exercises—actual modeling problems that focus on the concepts discussed in that chapter. Exercises appear as shown in the example below:



EXERCISE 3.2 VERIFYING LENS LAWS

The online code repository for chapter 3 contains a definition for a `Customer` entity and the lenses for it. Take a look at `addressLens`, which updates the address of a customer, and write properties using ScalaCheck that verify the laws of a lens.

Modeling a specific use case in the solution domain can be done in multiple ways. The exercises discuss these alternatives and the pros and cons of every approach. The reader is encouraged to try solving them independently before looking at the solutions available as part of the online repository of the book, which again can be found at the publisher's website (www.manning.com/books/functional-and-reactive-domain-modeling) and GitHub (<https://github.com/debasishg/frdomain>).

Author Online

Purchase of *Functional and Reactive Domain Modeling* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/books/functional-and-reactive-domain-modeling. This page provides information on how to get on the forum after you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It isn't a commitment to any specific amount of participation on the part of the author, whose contribution to the AO forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The AO forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author

Debasish Ghosh has been working on domain modeling for the last ten years and on functional modeling patterns for the last five. He has extensive experience with functional programming in his daily job, using languages like Scala and libraries like Scalaz and Akka, which are the cornerstones of this book. He has been one of the earliest adopters of event sourcing and CQRS and has implemented these techniques in real-world applications. Debasish is also the author of a related book on domain-specific languages, *DSLs in Action* (www.manning.com/books/dsls-in-action), published by Manning in 2010.

1

Functional domain modeling: an introduction

This chapter covers

- Domain models and domain-driven design
- Benefits of functional and pure domain models
- Reactive modeling for increased responsiveness
- How functional meets reactive

Suppose you're using the portal of a large online retail store to make purchases. After entering all the items, the shopping cart fails to register your purchases. How does that make you feel? Or say you do a price check on an item a week before Christmas, and the response comes back after an inordinate delay; do you like this shopping experience? In both cases, the applications weren't responsive. The first case depicts a lack of responsiveness to failure—your whole shopping cart went down because a back-end resource wasn't available. The second case illustrates a lack of responsiveness to varying load. Maybe it's the festive season that has triggered excessive load on the system and made your query response too slow. Both cases result in extreme frustration on the part of the user.

For any application that you develop, the core concept is the *domain model*, which is the representation of how the business works. For a banking system, the

system functionalities consist of entities such as banks, accounts, currency, transactions, and reporting that work together to deliver a good banking experience to the user. And it's the responsibility of the model to ensure that users have a good experience when they use the system.

When you implement a domain model, you translate the business processes into software. You try to make this translation in a way that results in the software resembling the original processes as much as possible. And to achieve this, you follow certain techniques and adopt paradigms in your design, development, and implementation. In this book, you'll explore how to use a combination of functional programming (FP) and reactive modeling to deliver models that are responsive and scalable, yet easy to manage and maintain. This chapter introduces fundamental concepts of these two paradigms and explains how the combination works together to achieve responsiveness of the model.

If you're designing and implementing systems right now, using any of the programming techniques that the industry currently offers, this book will open your eyes to new techniques for making your model more resilient and expressive. If you're managing teams that develop complex systems, you'll appreciate the benefits of using functional and reactive programming to deliver more reliable software for your clients. This book uses Scala as the implementation language, but the basic principles that you'll learn can be applied to many other languages used in the industry today.

Before you move on to the core topic of domain modeling, figure 1.1 shows how this chapter provides the groundwork for understanding the synergy of functional and reactive programming for implementing domain models. The idea is to make you comfortable with the basic concepts so that at the end of this chapter you'll be able to refine your domain-modeling techniques in terms of both functional and reactive paradigms.

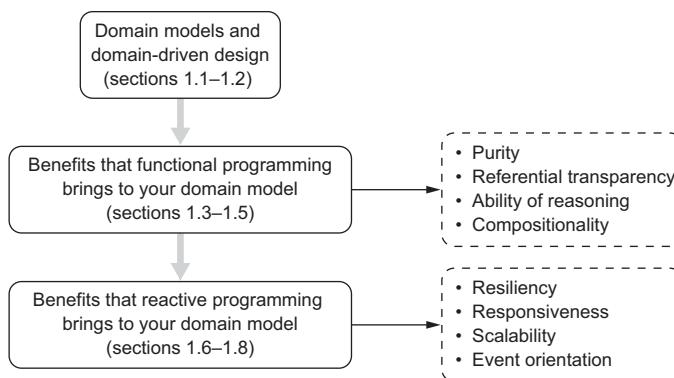


Figure 1.1 Building functional and reactive domain models

1.1 What is a domain model?

When was the last time you *withdrew* cash from an *ATM*? Or *deposited* cash into your *bank account*? Or used internet banking to check whether your monthly pay has been *credited* to your *checking account*? Or asked for a *portfolio statement* from your *bank*? All these italicized terms relate to the business of personal banking. We call this the *domain* of personal banking. The word *domain* here means the area of interest in the business. When you're developing a system to automate banking activities, you're modeling the business of personal banking. The abstractions that you design, the behaviors that you implement, and the UI interactions that you build all reflect the business of personal banking—together they constitute the *model* of the domain.

More formally, a domain model is a blueprint of the relationships between the various entities of the problem domain and sketches out other important details, such as the following:

- *Objects that belong to the domain*—For example, in the banking domain you have objects such as banks, accounts, and transactions.
- *Behaviors that those objects demonstrate in interacting among themselves*—For example, in a banking system you *debit* an account, and you *issue a statement* to your client. These are typical interactions that occur between the objects of your domain.
- *The language that the domain speaks*—When you're modeling the domain of personal banking, terms such as *debit*, *credit*, *portfolio*, and so on, or phrases such as “transfer 100 USD from account1 to account2,” occur quite ubiquitously and form the vocabulary of the domain.
- *The context within which the model operates*—This includes the set of assumptions and constraints that are relevant to the problem domain and are automatically applicable for the software model that you develop. A new bank account can be opened for a living person or entity only—this can be one of the assumptions that define a context of your domain model for personal banking.

As in any other modeling exercise, the most challenging aspect of implementing a domain model is managing its complexity. Some of these complexities are inherent to the problem, and you can't avoid them. These are called the *essential* complexities of the system. For example, when you apply for a personal loan from your bank, determining the eligibility of the amount depending on your profile has a fixed complexity that's determined by the core business rules of the domain. This is an essential complexity that you can't avoid in your solution model. But some complexities are introduced by the solution itself, such as when you implement a new banking solution that introduces extraneous load on operations in the form of additional batch processing. These are known as the *incidental* complexities of the model.

One of the essential aspects of an effective model implementation is reducing the amount of incidental complexity. And more often than not, you can reduce the incidental complexities of a model by adopting techniques that help you manage complexities better. For example, if your technique leads to better modularization of

your model, then your final implementation isn't a single monolithic, unmanageable piece of software. It's decomposed into multiple smaller components, each of which functions within its own context and assumptions. Figure 1.2 depicts such a system—I've shown two components for brevity. But you get the idea: With a modular system, each component is self-contained in functionality and interacts with other components only through explicitly defined contracts. With this arrangement, you can manage complexity better than with a monolithic system.

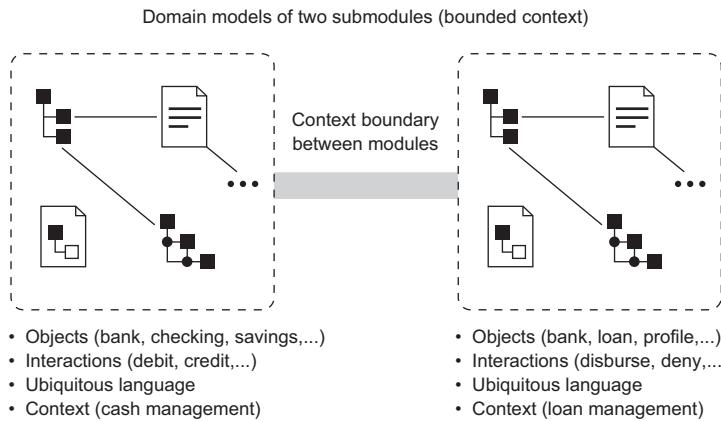


Figure 1.2 Overview of a domain model and its external context with terms from the personal banking domain. Each smaller module has its own set of assumptions and business rules, and these modules are easier to manage than a large monolithic system. But you need to keep the communication between them at a minimum and use explicitly defined protocols.

This book explains how adopting the principles of functional programming and combining them with a reactive design leads to the implementation of domain models that are easier to create, maintain, and use.

1.2 *Introducing domain-driven design*

In the previous section when explaining domain models, I used terms such as *banks*, *accounts*, *debit*, *credit*, and so forth. All of these terms are related to the personal banking domain and readily convey what roles they play in the functioning of a business. When you implement a domain model for personal banking, wouldn't it be convenient for users trying to understand your model if you used the same terminology as the business? For example, you may have as part of your model an entity named *Account* that implements all variations in behavior, depending on whether it's a checking, savings, or money market account. This is a direct mapping of concepts from the problem domain (the business) to the solution domain (your implementation).

When you're implementing a domain model, an understanding of the domain is of paramount importance. Only when you grasp how the various entities work in the real world will you have the knowledge to implement them as part of your solution. Understanding the domain and abstracting the central characteristics in the form of a model is known as *domain-driven design* (DDD). Eric Evans offers a wonderful treatment of the subject in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003).

1.2.1 **The bounded context**

Section 1.1 described modular models and a few advantages that modularization brings to a domain model. Any domain model of nontrivial complexity is really a collection of smaller models, each with its own data and domain vocabulary. In the world of domain-driven design, the term *bounded context* denotes one such smaller model within the whole. So the complete domain model is really a collection of bounded contexts. Let's consider a banking system: A portfolio management system, tax and regulatory reports, and term deposit management can be designed as separate bounded contexts. A bounded context is typically at a fairly high level of granularity and denotes one complete area of functionality within your system.

But when you have multiple bounded contexts in your complete domain model, how do you communicate between them? Remember, each bounded context is self-contained as a module but can have interactions with other bounded contexts. Typically, when you design your model, these communications are implemented as explicitly specified sets of services or interfaces. You'll see a few such implementations as we go along. The basic idea is to keep these interactions to the bare minimum so that each bounded context is cohesive enough within itself and yet loosely coupled with other bounded contexts.

You'll look at what's within each bounded context in the next section and learn about some of the fundamental domain-modeling elements that make up the guts of your model.

1.2.2 **The domain model elements**

Various kinds of abstractions define your domain model. If someone asks you to list a few elements from the personal banking domain, chances are you'll name items such as banks and accounts; account types such as checking, savings, and money market; and transaction types such as debit and credit. But you'll soon realize that many of these elements are similar with respect to how they're created, processed through the pipeline of the business, and ultimately evicted from the system. As an example, consider the lifecycle of a client account, as illustrated in figure 1.3. Every client account created by the bank passes through a set of states as a result of certain actions from the bank, the client, or any other external system.

Every account has an identity that has to be managed in the course of its entire lifetime within the system. We refer to such elements as *entities*. For an account, its identity is its account number. Many of its attributes may change during the lifetime of the system, but the account is always identified with the specific account number that was

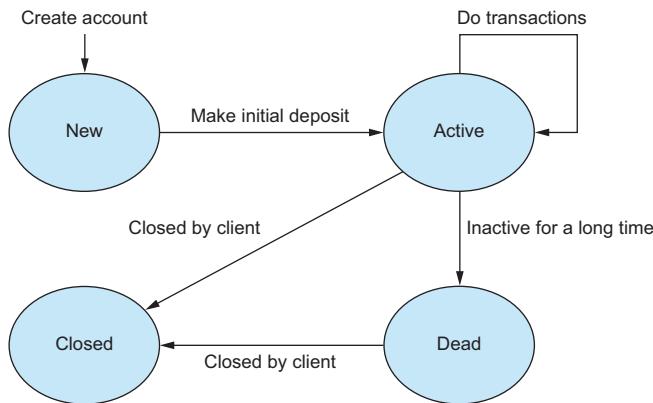


Figure 1.3 States in the lifecycle of a client account. Transition from one state to another depends on the action performed on the earlier state.

allocated to it when it was opened. Two accounts in the same name and having the same attributes are considered different entities because the account numbers differ.

Each account may have an address—the residential address of the account holder. An address is uniquely defined by the value that it contains. You change any attribute of an address, and it becomes a different address. Can you identify the difference in semantics between an account and an address? An address doesn't have any identity; it's identified entirely based on the value it contains. Not surprisingly, we call such objects *value objects*. Another way to distinguish between entities and value objects is that value objects are immutable—you can't change the contents of a value object without changing the object itself, after you create it.

The difference between an entity and a value object is one of the most fundamental concepts in domain modeling, and you must have a clear understanding of this. When we talk about an account, we mean a specific instance of the account, with an account number, the holder's name, and other attributes. Some of these attributes combined form a unique identity of the account. Typically, an account number is the identifying attribute of an account. Even if you have two accounts that have the same values for the nonidentifying attributes (such as the holder's name or the date it was opened), they are two different accounts if the account numbers are different. An account is an entity that has a specific identity, but with an address you need to consider only the value part. So within the model you can choose to have *only one instance* of a particular address and share it across all accounts that belong to the holders residing at that address. It's only the value that matters. You can change some of the attribute values in an entity, and yet the identity doesn't change; for example, you can change the address of an account, and yet it points to the same account. But you can't change the value of a value object; otherwise, it'll be a different value object. So a value object is immutable by definition.

Immutability semantics of entities and value objects

We'll have a different take on immutability of entities and value objects when we discuss implementations later in this chapter. In functional programming, our aim is to model as much immutability as possible—you'll model entities as immutable objects as well. So the best way to differentiate between entities and value objects is to remember that an entity has an *identity* that can't change, and a value object has a *value* that can't change. A value object is semantically immutable. An entity is semantically mutable, but you'll implement it using immutable constructs.

Is there any downside to modeling semantically mutable entities with immutable structures? Let's face it—mutable references are more performant. In many cases, working with immutable data structures leads to more objects being instantiated compared to using direct mutability, especially when a domain entity changes frequently. But as you'll see in this and the following chapters, mutable data structures lead to a fragile code base and make understanding code difficult in the face of concurrent operations. So the general advice is to start with immutable data structures—if you need to make some parts of the code more performant than what you get with immutability, go for mutation. But ensure that the client API doesn't get to see the mutation; encapsulate the mutation behind a referentially transparent wrapper function.¹

The heart of any domain model is the set of behaviors or interactions between the various domain elements. These behaviors are at a higher level of granularity than individual entities or value objects. We consider them to be the principal services that the model offers. Let's look at an example from the banking system. Say a customer comes to the bank or the ATM and transfers money between two accounts. This action results in a debit from one account and a credit to another, which will reflect as a change in balance in the respective accounts. Validation checks have to be done, determining, for instance, whether the accounts are active and whether the source account has enough funds to transfer. In every such interaction, many domain elements can be involved, both entities and value objects. In DDD, you model this entire set of behaviors as one or more *services*. Depending on the architecture and the specific bounded context of the model, you can package it as either a standalone service (you could name it `AccountService`) or as part of a collection of services in a more generic module named `BankingService`.

The main way that a domain service differs from an entity or a value object is in the level of granularity. In a service, multiple domain entities interact according to specific business rules and deliver a specific functionality in the system. From an implementation point of view, a service is a set of functions acting on a related set of domain entities and value objects. It encapsulates a complete business operation that has a

¹ As an example, look at the implementations of Scala's Collections API. Many of them, such as `List::take` or `List::drop`, use mutation under the hood, but the client API doesn't get to see it. The client gets back an immutable `List` for the call.

certain value to the user or the bank. Table 1.1 summarizes the characteristics of the three most important domain elements that you've seen so far.

Table 1.1 Domain elements

| Element | Characteristics |
|--------------|---|
| Entity | <ul style="list-style-type: none"> ▪ Has an identity ▪ Passes through multiple states in the lifecycle ▪ Usually has a definite lifecycle in the business |
| Value object | <ul style="list-style-type: none"> ▪ Semantically immutable ▪ Can be freely shared across entities |
| Service | <ul style="list-style-type: none"> ▪ More macro-level abstraction than entity or value object ▪ Involves multiple entities and value objects ▪ Usually models a use case of the business |

Figure 1.4 illustrates how the three types of domain elements are related in a sample from the personal banking domain. This is one of the fundamental concepts in DDD; make sure you understand the basics before continuing this journey.

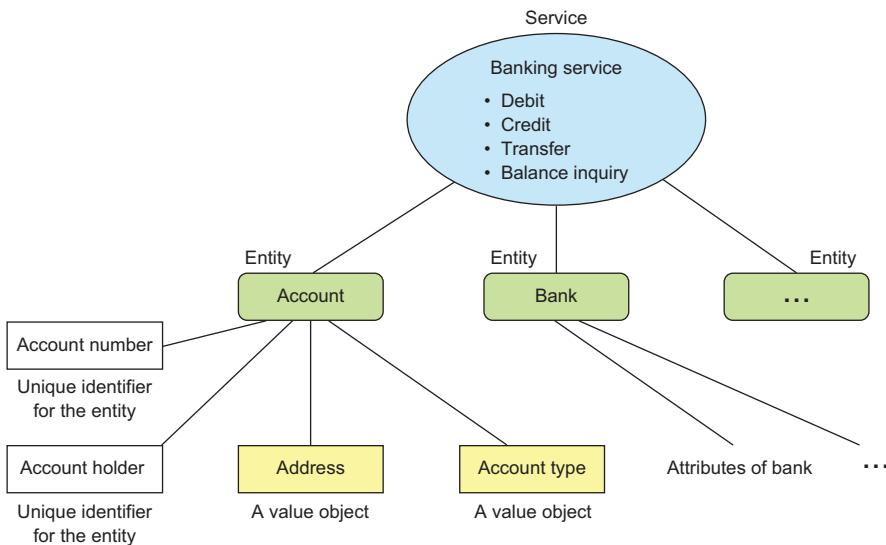


Figure 1.4 Relationships between the domain elements of a model. This example is from the personal banking domain. Note that Account, Bank, and so forth are entities. An entity can contain other entities or value objects. A service is at a higher level of granularity and implements behaviors that involve multiple domain elements.

DOMAIN ELEMENT SEMANTICS AND BOUNDED CONTEXT

Let's conclude this discussion on the various domain elements with an important concept that relates their semantics to the bounded context. When we say that an address is

a value object, it's a value object only within the scope of the bounded context in which it's being defined. In the bounded context of a personal banking application, an address may be a value object, and you don't need to track addresses by their identities. But consider another bounded context that implements a geocoding service. There you need to track addresses by latitude/longitude, and each address may have to be tagged with a unique ID. The address becomes an entity in this bounded context. Similarly, an account may be an entity in a personal banking application, whereas in a bounded context for portfolio reporting, you may have an account as a mere container of information that needs to be printed and hence implemented as a value object. The type of a domain element always reflects the bounded context where it's defined.

1.2.3 Lifecycle of a domain object

Every object (entity or value object) that you have in any model must have a definite lifecycle pattern. For every type of object you have in your model, you must have defined ways to handle each of the following events:

- *Creation*—How the object is created within the system. In the banking system, you may have a special abstraction that's responsible for creating bank accounts.
- *Participation in behaviors*—How the object is represented in memory when it interacts within the system. This is the way you model an entity or a value object within your system. A complex entity may consist of other entities as well as value objects. As an example, in figure 1.4, an Account entity may have references to other entities such as Bank or other value objects such as Address or Account Type.
- *Persistence*—How the object is maintained in the persistent form. This includes issues such as how you write the element to the persistent storage; how you retrieve the details in response to queries by the system; and if your persistent form is the relational database, how you insert, update, delete, or query an entity such as Account.

As always, a uniform vocabulary helps. The following section uses specific terms to refer to how we handle these three lifecycle events in our model. We call them *patterns* because we'll be using them repeatedly in various contexts of our domain modeling.²

FACTORIES

When you have a complex model, it's always a good practice to have dedicated abstractions that handle various parts of its lifecycle. Instead of littering the entire code base with snippets of the code that creates your entities, centralize them using a pattern. This strategy serves two purposes:

- It keeps all creational code in one place.
- It abstracts the process of creation of an entity from the caller.

² We use the term *patterns* in a slightly loose sense that may not conform exactly to the rigid structure that the GoF book follows (*Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et. al, Addison-Wesley Professional, 1994). We call *factories*, *repositories*, and *aggregates* domain lifecycle patterns, using the terminology of Eric Evans in his *Domain-Driven Design* book.

For example, you can have an account factory that takes the various parameters needed to create an account and hands you over a newly created account. The new account that you get back from the factory may be a checking, savings, or money market account depending on the parameters you pass. So the factory lets you create different types of objects using the same API. It abstracts the process and the type of created objects.

The creation logic resides within a factory. But where does the factory belong? A factory, after all, provides you with a service—the service of creation and possible initialization. It's the responsibility of the factory to hand you over a fully constructed, minimally valid instance of the domain object. One option is to make the factory part of the module that defines the domain object. This has a natural implementation in Scala using companion objects and is illustrated in the following listing. Another alternative is to think of a factory as part of a set of domain services. Chapter 2 details one such implementation.

Listing 1.1 Factory for instantiating accounts in Scala

```
trait Account {
    /**
     * ...
    */
}

case class CheckingAccount(/* parameters */) extends Account
case class SavingsAccount(/* parameters */) extends Account
case class MoneyMarketAccount(/* parameters */) extends Account

object Account {
    def apply(/* parameters */) = {
        // instantiate Checking, Savings or MoneyMarket account
        // depending on parameters
    }
}
```

Interface for Account entity and various types of accounts

Companion object in Scala that contains the factory

Factory method that instantiates accounts

AGGREGATES

In our model of personal banking, as you saw earlier, an Account can be thought of as composed of a group of related objects. Typically, this includes the following:

- Core account-identifying attributes such as the account number
- Various nonidentifying attributes such as holders' names, the date when the account was opened, and the date of closing (if it's a closed account)
- Reference to other objects, such as Address and Bank

One way you can visualize this entire graph of objects is to think of it as forming a *consistency boundary* within itself. When you have an account instantiated, all of these individual participating objects and attributes must be consistent as per the business rules of the domain. You can't have an account with a closing date preceding the opening date. And you can't have an account without any holders' names in it. These are all valid business rules, and the instantiated account must have *all* composing objects honor each of these rules. After you identify this set of participating objects in the

graph, this graph becomes an *aggregate*. An aggregate can consist of one or more entities and value objects (and other primitive attributes). Besides ensuring the consistency of business rules, an aggregate within a bounded context is also often looked at as a transaction boundary in the model.

One of the entities within the aggregate forms the *aggregate root*. It's sort of the guardian of the entire graph and serves as the single point of interaction of the aggregate with its clients. The aggregate root has two objectives to enforce:

- Ensure the consistency boundary of business rules and transactions within the aggregate.
- Prevent the implementation of the aggregate from leaking out to its clients, acting as a façade for all the operations that the aggregate supports.

Listing 1.2 shows the design of an Account aggregate in Scala. It consists of the aggregate root `Account` (which is also an entity), and has entities such as `Bank` and value objects such as `Address` as part of its composing elements.³ Designing aggregates isn't an easy task, especially if you're a beginner in DDD. Besides Eric Evans' *Domain-Driven Design* book, take a look at the "Effective Aggregate Design" article from Vaughn Vernon that discusses in three parts the various considerations that you need to make to design a good aggregate (http://dddcommunity.org/library/vernon_2011/).

Listing 1.2 Account as an aggregate

```
trait Account {
    def no: String
    def name: String
    def bank: Bank
    def address: Address
    def dateOfOpening: Date,
    def dateOfClose: Option[Date]
    //...
}

case class CheckingAccount (
    no: String,
    name: String,
    bank: Bank,
    address: Address,
    dateOfOpening: Date,
    dateOfClose: Option[Date],
    //...
) extends Account
```

The code listing is annotated with several callout boxes and arrows pointing to specific parts of the code:

- A box labeled "Basic contract of an Account aggregate" points to the `trait Account` definition.
- A box labeled "Reference to another entity" points to the `bank` field.
- A box labeled "Address is a value object." points to the `Address` type used in the `address` field.
- A box labeled "A concrete implementation of Account. Note that the fields override the defs of the trait." points to the `CheckingAccount` class definition.

³ In reality, when you design aggregates, you may find that for performance and consistency of operations you have to optimize away many composing entities from the aggregate and have only the root along with the value objects. For example, you may choose to keep a bank ID instead of the entire `Bank` entity as part of the `Account` aggregate.

```

case class SavingsAccount(
    /**
     * ...
     */
    rateOfInterest: BigDecimal,
    /**
     * ...
     */
) extends Account

trait AccountService {
    def transfer(from: Account, to: Account, amount: Amount): Option[Amount]
}

```

Case classes in Scala

Listing 1.2 uses Scala's case classes to model an `Account` aggregate. Case classes in Scala provide a convenient way to design objects that offer immutability from the ground up. All parameters that the class takes are immutable by default. Therefore, using case classes, we get the convenience of an easy-to-use way to define an aggregate, as well as all the benefits of immutability built right into it.

Listings 1.1 and 1.2 use traits in Scala. *Traits* enable you to define modules in Scala that can be composed together using mixin-based composition. *Mixins* are small abstractions that can be mixed in with other components to form larger components.

For more details on case classes, mixins, and traits, take a look at the official Scala home page (www.scala-lang.org) or the book *Programming in Scala* by Martin Odersky et al. 3rd Ed. (Artima Press, 2016).

Note that we've implemented the basic contract of an `Account` aggregate as a trait in Scala and the variants in the form of case classes. As the preceding sidebar "Case classes in Scala" indicates, case classes are conveniently used to model immutable data structures. These are known as *algebraic data types*, which we'll discuss in more detail as we move along. But let's look at one aspect of the account entity aggregate touched on previously.

In section 1.2.2, you saw that you can update some attributes of an entity without changing its identity. This means an entity should be updateable. But here we've modeled the entity `Account` as an *immutable* abstraction. Does this seem like an apparent contradiction? Absolutely not! We'll allow updating of entities, but in a functional way that doesn't make the entity mutable *in place*. Instead of mutating the object itself, your updates will produce a new instance with the modified attribute values.⁴ This has the advantage that you can still continue sharing your original abstraction as an immutable entity while doing updates that generate new instances of the same entity. In the context of the functional way of thinking, you'll strive for immutability of entities (much like value objects) as much as you can. And this is one of the guiding

⁴ If you're impatient, jump to listing 1.4, where the `debit` and `credit` methods create new instances of `Account` with the updated balance amount.

principles that will dictate your model design. Listing 1.2 also shows an example domain service (`AccountService`) that uses the `Account` aggregate to implement a transfer of funds between two accounts.

REPOSITORIES

As you know, aggregates are created by factories and represent the underlying entities in memory during the active phase of the objects' lifecycle (see figure 1.3 to review the lifecycle of an account). But you also need a way to persist an aggregate when you no longer need it. You can't throw it away, because you may need to fetch it later for another purpose.

A *repository* gives you this interface for parking an aggregate in a persistent form so that you can fetch it back to an in-memory entity representation when you need it. Usually a repository has an implementation based on persistent storage such as a relational database management system (RDBMS), though the contract doesn't enforce that.⁵ Also note that the persistent model of the aggregate may be entirely different from the in-memory aggregate representation and is mostly driven by the underlying storage data model. It's the responsibility of the repository (see the following listing) to provide the interface for manipulating entities from the persistent storage without exposing the underlying relational (or whatever model the underlying storage supports) data model.

Listing 1.3 AccountRepository—interface for manipulating accounts from the database

```
trait AccountRepository {  
    def query(accountNo: String): Option[Account]  
    def query(criteria: Criteria[Account]): Seq[Account]  
    def write(accounts: Seq[Account]): Boolean  
    def delete(account: Account): Boolean  
}
```

The interface for a repository doesn't have any knowledge of the nature of the underlying persistent store. It can be a relational database or a NoSQL database—only the implementation knows that. So what an aggregate offers for in-memory representation of the entity, a repository does the same for the persistent storage. An aggregate hides the underlying details of the in-memory representation of the object, whereas a repository abstracts the underlying details of the persistent representation of the object. Listing 1.3 shows an `AccountRepository` for manipulating accounts from the underlying storage; the listing doesn't show any specific implementation of the repository. But the user still interacts with the repository through an aggregate. Look at this

⁵ In many small applications, you can have an in-memory repository. But that's not usually the case.

sequence to get an idea of how an aggregate provides a single window to the entire lifecycle of an entity:

- You supply a bunch of arguments to the factory and get back an aggregate (such as `Account`).
- You use the aggregate (`Account` in listing 1.2) as your contract through all behaviors that you implement through services (`AccountService` in listing 1.2).
- You use the aggregate to persist the entity in the repository (`AccountRepository` in listing 1.3).

So far you've seen modularization in models using the bounded context, the three most important types of domain elements that you need to implement (entities, value objects, and services), and the three patterns used to manipulate them (factories, aggregates, and repositories). As you must have realized by now, the three types of elements participate in domain interactions (such as debit, credit, and so forth in the banking system), and their lifecycles are controlled by the three patterns. The last thing to discuss in domain-driven design is an aspect that binds all of them together. It's called the *vocabulary* of the model, and in the next section you'll learn why it's important.

1.2.4 The ubiquitous language

Now you have the entities, value objects, and services that form the model, and you know that all these elements need to interact with each other to implement the various behaviors that the business executes. As a software craftsperson, it's your responsibility to model this interaction in such a way that it's understandable not only to the hardware underneath, but also to a curious human mind. This interaction needs to reflect the underlying business semantics and must contain vocabulary from the problem domain you're modeling. By *vocabulary*, I mean the names of participating objects and the behaviors that are executed as part of the use cases. In our example, entities such as `Bank`, `Account`, `Customer`, and `Balance` and behaviors like `debit` and `credit` resonate strongly with the terms of the business and hence form part of the domain vocabulary. The use of domain vocabulary needs to be transitively extended to larger abstractions formed out of smaller ones. For example, you can compose an `AccountService` implementation (which is a domain service) as follows:⁶

```
trait AccountService {
    def debit(a: Account, amount: Amount): Try[Account] = //...
    def credit(a: Account, amount: Amount): Try[Account] = //...

    def transfer(from: Account, to: Account, amount: Amount) = for {
        d <- debit(from, amount)
        c <- credit(to, amount)
    } yield (d, c)
}
```

⁶ If you don't understand the details of the implementation, it's okay. You'll implement this service later in this chapter as we discuss the evolution of functional domain models.

Let's take a more detailed look at what this implementation exemplifies with respect to the qualities of understandability just described:

- The function body is minimal and doesn't contain any irrelevant details. It just encapsulates the domain logic involved in a transfer of funds between two accounts.
- The implementation uses terms from the domain of banking, so a person familiar with the business domain who doesn't know anything about the underlying implementation platform should also be able to understand what's going on.
- The implementation narrates just the happy path of execution. The exceptional paths are completely encapsulated within the abstractions that you use for implementation. In case you know Scala, the for-comprehension used here is monadic and takes care of any exceptions that may happen in the sequence of execution.⁷ We'll discuss many of these as we move along.

Eric Evans calls this the *ubiquitous language*. Use the domain vocabulary in your model and make the terms interact in such a way that it resembles the language that the domain speaks. Start with the correct naming of entities and atomic behaviors, and extend this vocabulary to larger abstractions that you compose out of them. Different modules can speak different dialects of the language, and the same term may mean something different in a different bounded context. But within the context, the vocabulary should be clear and unambiguous.

Having a consistent ubiquitous language has a lot to do with designing proper APIs of your model. The APIs must be expressive so that a person who's an expert in the domain can understand the context by looking at the API only. This is known as *domain-specific language*; see my book *DSLs in Action* (Manning, 2010) for a detailed treatment of the subject.

1.3 Thinking functionally

Many approaches to domain modeling exist, but over the last decade or so, object-oriented (OO) technologies have completely dominated in carving out the most complex of domain models. In this book, I'm going to be a bit radical and use plain old *functions* as the main abstraction to model domain behaviors. Over the next couple of sections, you'll see the benefits of doing that, from the perspective of both modeling and maintaining your software.

Sometimes, the elegant implementation is just a function. Not a method.
Not a class. Not a framework. Just a function.

—John Carmack on Twitter
(https://twitter.com/ID_AA_Carmack/statuses/53512300451201024)

⁷ In Scala, a for-comprehension is syntactic sugar for chaining together map/flatMap/filter operations. For more details, see <http://docs.scala-lang.org/tutorials/tour/sequence-comprehensions.html>.

But let's start with the paradigm that we've all been using over the past few years. In this example, you'll dissect an implementation and gradually morph it into a functional variant. Along the way, I'll highlight the benefits that the latter will bring you. Let's go back to a domain that we all interact with in our daily lives: personal banking. You'll consider a simple model consisting of an aggregate, Account, which has a value object, Balance, a few other attributes, and a couple of operations for debiting and crediting from and to the account, as shown in the following listing.

Listing 1.4 Sample model from the personal banking domain

```

type Amount = BigDecimal

case class Balance(amount: Amount = 0)                                Aggregate

class Account(val no: String, val name: String, val dateOfOpening: Date) { ←
    var balance: Balance = Balance()                                     ←

    def debit(a: Amount) = {                                              ←
        if (balance.amount < a)
            throw new Exception("Insufficient balance in account")
        balance = Balance(balance.amount - a)
    }                                                               ←

    def credit(a: Amount) = balance = Balance(balance.amount + a)      ←
}

val a = new Account("a1", "John")                                       ←
a.balance == Balance(0)                                                 ←

a.credit(100)                                                       ←
a.balance == Balance(100)                                               ←

a.debit(20)                                                        ←
a.balance == Balance(80)                                              ←

```

Operations that mutate state: A callout box pointing to the `debit` and `credit` methods, indicating they directly modify the `balance` field.

Aggregate: A callout box pointing to the `Account` class, identifying it as an aggregate.

Mutable state in aggregate: A callout box pointing to the `balance` field within the `Account` class definition.

An example assertion that returns true if equality is satisfied: A callout box pointing to the check for equality in the test code (`a.balance == Balance(100)`).

Listing 1.4 is self-explanatory. The class `Account` holds a mutable state, which is the balance that the account holds. The methods `debit` and `credit` directly mutate the state of the object to change the balance amount that the account holds at any point in time.



QUIZ TIME 1.1 What do you think is the primary drawback of this model?

Think for a moment. Take a second look at listing 1.4 if you need to. What we're going to discuss is possibly one of the most important reasons why you should appreciate the functional way of thinking and modeling.



QUIZ MASTER'S RESPONSE 1.1 It's the *mutability* that hits you in two ways: It makes it hard to use the abstraction in a concurrent setting and makes it difficult to reason about your code.

Here's a slightly longer explanation. The var balance: Balance is the mutable state in our domain model. The key word here is *mutable*, which indicates that the state (balance), which this object holds, can be updated in place by multiple clients of the objects. This can lead to issues in a concurrent environment, where you can have various types of inconsistencies in determining the value of the state at any point in time. This is a huge topic in itself, and you can get a clearer picture of all such issues from the excellent book *Java Concurrency in Practice* by Brian Goetz (Addison-Wesley Professional, 2006). Mutable state is also an antipattern when it comes to reasoning about your code, as you'll see later in this chapter.⁸ Though it appears to be a convincing way of modeling the world, mutable states create more problems than solutions. You need to find a way to get rid of these mutable states.

Let's see if we can improve upon the primary drawback of the previous code and stay within the realm of object-oriented thinking. The following listing shows our next attempt toward purification of the sin committed in listing 1.4 by introducing a mutable state in our model.

Listing 1.5 Immutable Account model

```
type Amount = BigDecimal

case class Balance(amount: Amount = 0)

class Account(val no: String, val name: String,
    val dateOfOpening: Date, val balance: Balance = Balance()) { ←
    def debit(a: Amount) = {
        if (balance.amount < a)
            throw new Exception("Insufficient balance in account")
        new Account(no, name, dateOfOpening, Balance(balance.amount - a))
    }
    def credit(a: Amount) =
        new Account(no, name, dateOfOpening, Balance(balance.amount + a))
}

val a = new Account("a1", "John", today)
a.balance == Balance(0) ←
val b = a.credit(100)
a.balance == Balance(0)
b.balance == Balance(100) ←
val c = b.debit(20)
b.balance == Balance(100)
c.balance == Balance(80) ←
```

Balance is now immutable.

The operations debit and credit create new instances of Account.

Immutability in action—account balance isn't mutated in place.

The mutable state is gone! Every operation on Account creates a new object with the *modified state*. Instead of having a mutable state, the new Account class carries the state with itself. Once you have an instance of the class, you have the balance as a state within

⁸ If the phrase *reasoning about your code* sounds unfamiliar, don't worry. You'll learn more about it shortly.

itself. But the difference is that this state is immutable. You can't change its value without creating another `Account` object. And that's exactly what our `debit` and `credit` operations do here. Scala ensures that the parameter you pass in a class constructor is immutable by default. You can choose to make it mutable by making it a `var`. But that's an explicit modifier that you need to apply to get mutability—another excellent decision that encourages immutable abstraction design.

Now that you've made `Account` an immutable abstraction, you can freely share `Account` across threads in a concurrent setting.⁹ This is a huge gain and is your first baby step toward appreciating the virtues of functional thinking, which works in terms of pure functions that accept input and generate output without relying on or impacting any shared mutable state. Immutability has a big role to play here.

But you're not finished yet. `Account` is still an abstraction that holds both the state and the behavior. The idea is to decouple the two, which, as you'll see later, will give you better modularity and hence better compositionality. But first let's look at the virtues that your code will have if you can model it using pure functions.

1.3.1 Ah, the joys of purity

Imagine you've gone back to your school days and are trying to learn the definition of a *function* from the mathematical point of view. Because we're discussing functional programming here, how different is this function from the one you learned in math class?

In mathematics, a function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output.

—Wikipedia, [http://en.wikipedia.org/wiki/Function_\(mathematics\)](http://en.wikipedia.org/wiki/Function_(mathematics))

This definition never mentions dependency of a function on shared mutable states. The output of the function is purely determined from the inputs—much like figure 1.5, which models a function (f) as a black box that transforms an input (x) to an output (y). In functional programming, you strive to make your functions behave like a mathematical function only.

 **QUIZ TIME 1.2** Which model, listing 1.4 or listing 1.5, looks closer to the definition of *function* just introduced in this section?

Now that you've seen the definition of a function and understand how you should try to achieve the same effect in your domain models, this question should be a piece of cake. The lesser the dependency on external mutable state, the closer the model is to the purity of a mathematical function.

⁹ Here I'm talking only about sharing `Account` objects. You still need to manage atomicity if you want to compose multiple debits and credits within a single transaction.



QUIZ MASTER'S RESPONSE 1.2 Listing 1.5, which makes Account an immutable abstraction, is closer to this definition.

In our $y = f(x)$ model in figure 1.5, assume f is the function square. Then $\text{square}(3) = 9$ and the result will be exactly the same regardless of how many times you invoke f . Let's discuss this aspect in more detail with the two Account models just introduced.

In the mutable model shown in listing 1.4, invoking a `debit(100)` on an `Account` object yields an amount that depends not only on the input parameter, 100, and the object itself (which you can consider an implicit parameter as well), but also on other clients *sharing* the same object. This is because all clients sharing the `Account` object have equal access to the mutable state. This is far from what we discussed as a pure function in this section.

In the immutable model shown in listing 1.5, the `Account` object itself also holds the current state. Therefore, invoking `debit(100)` on an `Account` object holding the current balance amount of 2000 will always yield a new `Account` object with an updated balance of amount 1900. The output depends only on the input being supplied. So this model has the purity of a mathematical function.

Okay, now it's time to reveal the oracle. The immutable model of `Account` is an object-oriented version of the functional model that you'll see soon. It still has the functions modeled as methods of a class. With this approach, you're often faced with the dilemma of which function should be part of which class. Also, it becomes difficult to compose functions implemented as methods of different classes.

In this example, `debit` and `credit` are operations on a single account, and you've kept them as behaviors of `Account`. But an operation such as `transfer` has two accounts. Should it also be part of the `Account` class, or should you have it as part of a domain service? How should you deal with other services on an account, such as the daily balance statement or interest calculation? You may tend to put them within one class and make it a bloated abstraction. Putting such behaviors within a specific aggregate also hampers modularity and compositionality. Here are the general principles you need to follow when designing functional domain models:

- Model the immutable state in an *algebraic data type* (ADT).
- Model behaviors as functions in modules, where a module represents a coarse unit of business functionality (for example, a domain service). This way, you separate state from behavior. Behaviors compose better than states; therefore, keeping related behaviors in modules enables more compositionality.
- Keep in mind that behaviors in modules operate on types that the ADTs represent.

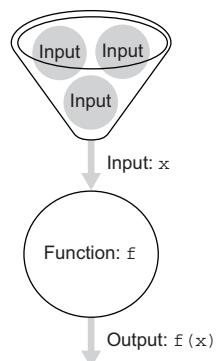


Figure 1.5 $y = f(x)$ models a pure function. f is a black box that transforms an input, x , into an output, y .



QUIZ TIME 1.3 True or false: The object-oriented paradigm¹⁰ couples state and behavior. Functional programming decouples them.

Let's first take a look at implementing our Account model using functional Scala. Then you'll be able to answer the quiz. Listing 1.6 is the model that improves the earlier implementation. It contains quite a few Scala constructs, some of which you can ignore for the time being. But here are the main points for your domain model that come from functional thinking:

- Case class models an ADT in Scala. By default, all parameters of the ADT are immutable, which implies that you don't need any special machinery to ensure the immutability of your model.
- The definition of the ADT doesn't contain any behavior. Note that debit and credit are now within AccountService, which you've defined as a domain service.¹¹ Services are defined in modules, which are implemented as traits in Scala. Traits act as mixins and enable easy composition to form larger modules out of smaller ones. When you need a concrete instance of a module (a service in our context), you use the object keyword. As I mentioned earlier, with functional thinking you decouple state from behavior—the state now resides within the ADT, and the behaviors are modeled as standalone functions within modules.
- debit and credit are pure functions because they aren't tied to any specific object. Instead they take arguments, perform some functionality, and generate specific outputs, just like our $y = f(x)$ model in figure 1.5.
- Listing 1.6 uses a few other constructs such as Try, Success, and Failure that are more functional and compositional than throwing exceptions. The upcoming sidebar “Exceptions in Scala” gives an overview of handling exceptions functionally in Scala. Later chapters also cover this topic, as they detail functional programming patterns.

Listing 1.6 Purifying the model

```
import java.util.{ Date, Calendar }
import scala.util.{ Try, Success, Failure }

def today = Calendar.getInstance.getTime
type Amount = BigDecimal

case class Balance(amount: Amount = 0)

case class Account(no: String, name: String,
  dateOfOpening: Date, balance: Balance = Balance())
```

← **Account aggregate
is now an ADT**

¹⁰ As implemented in major, mainstream OO languages.

¹¹ If you've forgotten about domain services, see section 1.2.3.

```

trait AccountService {

    def debit(a: Account, amount: Amount): Try[Account] = {
        if (a.balance.amount < amount)
            Failure(new Exception("Insufficient balance in account"))
        else Success(a.copy(balance = Balance(a.balance.amount - amount)))
    }

    def credit(a: Account, amount: Amount): Try[Account] =
        Success(a.copy(balance = Balance(a.balance.amount + amount)))
}

object AccountService extends AccountService
import AccountService._

val a = Account("a1", "John", today)
a.balance == Balance(0)

val b = credit(a, 1000)

```

Domain service with the operations debit and credit

Concrete instance of the service using the object keyword

Pure function invocation: returns a Try[Account]

Figure 1.6 summarizes the changes that morph our object-oriented, immutable domain model into the functional variant using Scala.

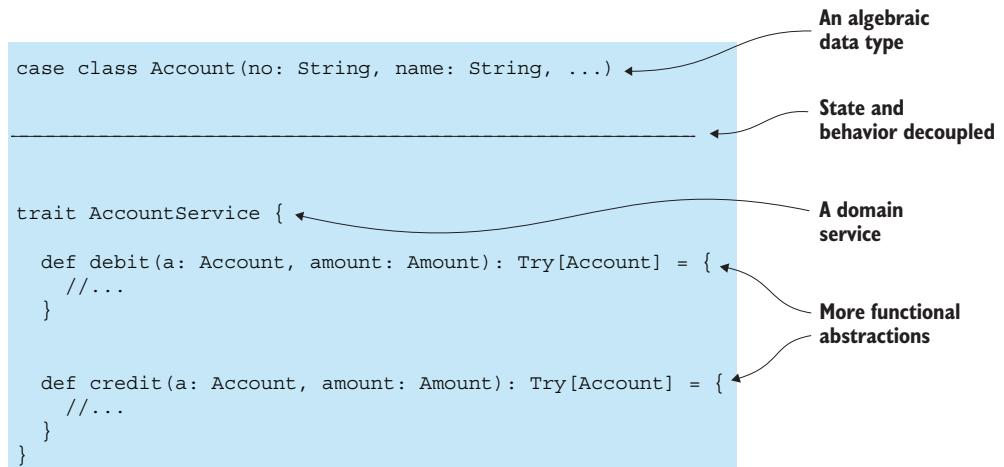


Figure 1.6 From object-oriented immutable modeling to functional abstractions. Note that we've separated the state from the behavior. The state is encoded within an algebraic data type, `Account`, whereas the behaviors are within a domain service. Also, constructs such as `Try` help in building compositional abstractions.



QUIZ MASTER'S RESPONSE 1.3 The mainstream object-oriented languages encourage functions to be encapsulated under the same abstraction as the state. In class-oriented OO languages this abstraction is the “class.”

The next section covers function composition. But let me give you a sneak peek at another cool compositional effect that results from your refactoring into functional abstractions such as Try. You can now compose multiple debits and credits as follows:

```
val a = Account("a1", "John", today)

for {
  b <- credit(a, 1000)
  c <- debit(b, 200)
  d <- debit(c, 190)
} yield d
res5: scala.util.Try[Account] = Success(Account(a1, John, Sat Nov 22
02:38:03 GMT+05:30 2014, Balance(610)))
```

Exceptions in Scala

Exceptions are considered impure in functional programming. To handle exceptions functionally, Scala defines an abstraction, `util.Try`, with two concrete implementations for `Success` and `Failure`. Listing 1.6 uses this abstraction to handle any exception that may arise from the `generateAuditLog` operation. Note that `generateAuditLog` is a function that takes an account and an amount and tries to generate an audit log, which is a string. `Try[String]` as the return type publishes the fact that this operation can fail and in that case will return a `Failure`. It's not essential to understand the details now. But be aware that `Try` is a composable abstraction and can be combined with the other abstractions in a pure and functional way.

1.3.2 Pure functions compose

What is *function composition*? Before you answer that, let's check the definition of *composition*:

The way in which something is put together or arranged: the combination of parts or elements that make up something

—Merriam Webster (www.merriam-webster.com/dictionary/composition)

When you *compose*, you combine parts to make a whole. In mathematics, it's the pointwise application of one function to another to produce a third function. Clearly there's an implication of creativity—you create a new function that combines the effect of two functions. For example, the two functions $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ can be composed together to get a new function that maps every x in X to $g(f(x))$ in Z .

Let's now translate this example to the domain of functional programming. Suppose you have a function, `square: Int -> Int`, which takes an integer as an argument and produces another integer, its value squared, as output. And you have another function, `add2: Int -> Int`, which takes an integer and adds 2 to it. You define the composition of these two functions as `add2(square(x: Int))`, which adds 2 to the square of the integer that you give as input to the composed function.

This is the first step toward appreciating the fact that functional programming is based on function composition, which in turn is similar to the way we treat functions in mathematics. This is known as the *compositionality property* of functional programs.



QUIZ TIME 1.4 Suppose you have two functions: $f: \text{String} \rightarrow \text{Int}$ and $g: \text{Int} \rightarrow \text{Int}$. How do you define the composition of f and g ? Can you think of some real functions that satisfy these signatures?

Using the property of compositionality, you can build bigger functions out of smaller ones. One of the main themes of this book is the exploration of various ways you can make functions compose. We'll be using Scala, which offers capabilities that make such composition easy. For a detailed treatment of functional programming in Scala, check out Paul Chiusano and Runar Bjarnason's excellent book, *Functional Programming in Scala* (Manning, 2014).

You'll look at examples of composing functions in Scala using the Scala REPL, which is the environment for interacting with the Scala interpreter. But first, the quiz master is ready with the answer to the previous quiz.



QUIZ MASTER'S RESPONSE 1.4 The composition is defined as $g(f(x: \text{String}))$. A realistic example is to treat f as a function that computes the length of a string, and g as a function that doubles the input integer. So $\text{double}(\text{length}(x: \text{String}))$ is a practical example of composing the two functions that returns twice the length of the input string.

Now that you're familiar with the basic technique of function composition, it's time to take the next step. So far in discussing composition, I've referred to individual functions pipelined together, with one of them receiving as input what the other function generated. But when we talk about compositionality properties in functional programming, it's much more than that. Let's look at the example in figure 1.7.

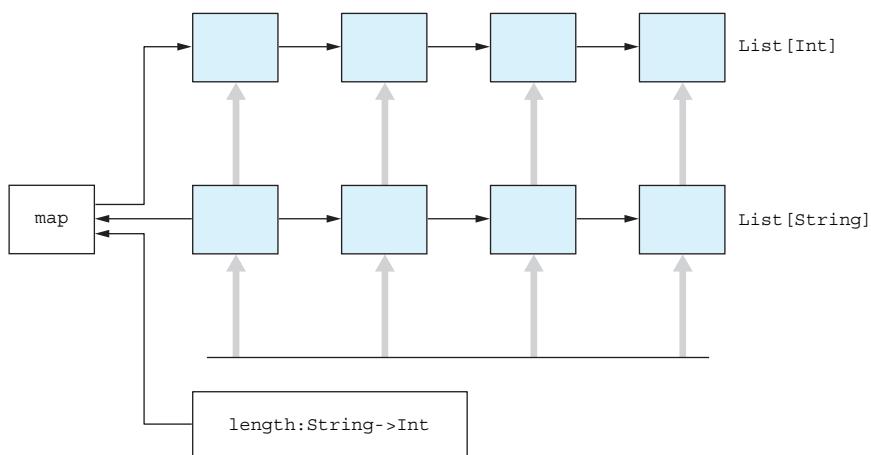


Figure 1.7 `map` is a higher-order function that takes another function as input.

The function `map` takes two parameters—a list of strings and another function, `length: String -> Int`. `map` iterates over the list, and applies the function `length` to every element of the list. The result it generates is another list, and each of those elements is the result of applying `length`, which is a list of integers. This is an excellent example of how to think functionally and points to some interesting characteristics of this paradigm, as table 1.2 shows. Higher-order functions such as `map` are also known as *combinators*.

Table 1.2 Thinking functionally with the `map` function

| Characteristics of the <code>map</code> function | How it relates to functional programming |
|--|--|
| <p>You can pass a function as an argument. In our example, <code>map</code> takes a function <code>length</code>.</p> <p><code>map</code> is a function that takes another function as input.</p> <p><code>map</code> iterates through the list of strings, but the looping is abstracted from the API user.</p> | <p>Functions are first-class abstractions.</p> <p><code>map</code> is a higher-order function.</p> <p>With functional programming, you tell the function <i>what</i> to do. <i>How</i> it will be done is abstracted from the API user. You can have <code>map</code> for other types of sequences as well (not only a list), and the iteration is handled by the <code>map</code> implementation.</p> |

 **QUIZ TIME 1.5** What happens if the function that you pass on to `map` also happens to update a shared mutable state? Does this mean iterating a list multiple times will lead to different outputs?

The code in the following listing uses higher-order functions such as `map` in Scala to demonstrate several ways to compose. Each example follows the guiding principles of functional thinking shown in table 1.2.

Listing 1.7 Function composition and higher-order functions

```
scala> val inc1 = (n: Int) => n + 1
inc1: Int => Int = <function1>
← Function that adds 1
to an integer

scala> val square = (n: Int) => n * n
square: Int => Int = <function1>
← Function that squares
its integer input

scala> (1 to 10) map inc1
res1: scala.collection.immutable.IndexedSeq[Int] =
→ Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
← Map the increment function
over a collection.

scala> (1 to 10) map square
res4: scala.collection.immutable.IndexedSeq[Int] =
→ Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
← Map the square function
over a collection.

scala> val incNSquare = inc1 andThen square
incNSquare: Int => Int = <function1>
← Define a function composition
(add 1, then square).
```

```

scala> incNSquare(4)
res6: Int = 25

scala> val squareNInc = inc1 compose square
squareNInc: Int => Int = <function1>

scala> squareNInc(4)
res8: Int = 17

```

Another way to define
composition (square,
then add 1)

Now it's time to use some of these combinators to enrich our Balance domain model. In complex domain modeling, you'll define lots of combinators on your own. But the ones that come with the standard library are extremely useful, and you'll often find yourself falling back on many of them while building your own. After all, it's compositionality that you're after.



QUIZ MASTER'S RESPONSE 1.5 The quiz master is firm on this. Don't violate purity when it comes to combinators such as `map` and others. The function that you pass to `map` must be free of any side effects or mutation. We'll talk about side effects shortly.

Now let's try to implement some domain behavior in our personal banking system. Suppose you want to add an auditing capability to transactions (such as debit and credit) that generates audit logs and writes them somewhere. The details are omitted in the example—they aren't important for understanding the concept at hand. Assume you have the following two functions:

- `generateAuditLog: (Account, Amount) => Try[String]`
- `write: String => Unit`

This would have been a simple exercise with an imperative programming model. But here the idea is to use function composition and higher-order functions to achieve the same result. The following listing illustrates the implementation.

Listing 1.8 Composition through higher-order functions

```

val generateAuditLog: (Account, Amount) => Try[String] = //...
val write: String => Unit

debit(source, amount)
  .flatMap(b => generateAuditLog(b, amount))
  .foreach(write)

```

Debit the account. Debit
returns a `Try[Account]`,
which you `flatMap` in #B.

If log generated,
write to database

If debit is okay,
generate audit log

The requirement is to define a function that performs the following sequence of operations:

- 1 Debit an account.
- 2 If the debit goes through, generate an audit log; otherwise, stop.
- 3 Write the log to a store.

You need to implement exactly this sequence by using the compositionality of combinators that functional programming offers. The flow of activities in this sequence and, ideally, the modeling of your domain behaviors should reflect the same sequence. Thanks to all the functional thinking that you've done so far and all the combinators discussed earlier, listing 1.8 provides a faithful portrayal of this same logic.

The following sequence describes the flow of actions in listing 1.8. And if you're like me and love to see such interactions in the form of a diagram, check out figure 1.8.

- 1 The call to debit can generate a Failure (with an exception) or a successful generation of the modified Account.
- 2 In case of failure, the entire sequence is broken and you're finished. There's no explicit check for failure here. All such boilerplates are hidden behind the implementation of the map combinator.
- 3 If debit generates a successful Account, this value is threaded into the flatMap combinator and passes on to the function generateAuditLog.
- 4 generateAuditLog is again a pure function that generates a String as the log line, which gets fed into foreach. If the generation of the log line fails, you're finished and the sequence is broken.
- 5 foreach is a combinator used for side-effecting operations. The last stage of the pipeline is to write the log record into the database or filesystem, which is necessarily a side effect. You use foreach to implement that.

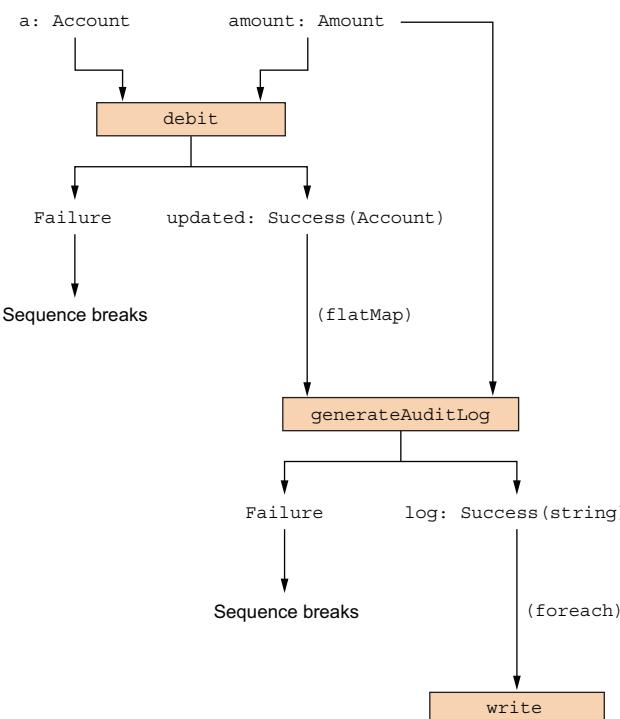


Figure 1.8 Enrich domain behavior through function composition using combinators. Note the Success path and the Failure path in the flow. A failure breaks the sequence immediately.

The primary takeaway of this section is to appreciate how function composition through combinators can lead to enrichment of domain behaviors. Composing smaller combinators to yield larger behaviors and functional thinking is the way to achieve that. In the next section, you'll learn how functional thinking can lead to code that you can reason about, much like functions in mathematics.

1.4 Managing side effects

So far, we've discussed a lot of properties that pure functions have that make them compositional and that allow you to design beautiful abstractions for your domain. If the reality was so simple and all functions that you write for your domain model were so pure, then the entire industry of software development wouldn't have seen so many failed projects.

Consider a simple function, `square`, that takes an integer as input and outputs the square value. But unlike the previous one, in addition to outputting the square of the integer, it has to write the result in a specific file on the filesystem. The rules of the game have changed now. What happens if you get an I/O exception when trying to write the output in the specific file? Maybe the disk is full. Or you don't have write permission in the folder where you're trying to create the output file.

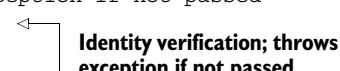
All of these are valid concerns, and your code needs to take care of handling all these exceptions—which means your function now has to deal with external entities (such as the filesystem) that aren't part of the explicit input that it receives. The function is no longer *pure*. It has to deal with external effects that belong to the outside world, which are known as *side effects*.

 **QUIZ TIME 1.6** You've already seen examples of side effects in this chapter. Can you point out where?

It's not that side effects are evil; they're one of the essential components that you need to manage when designing a nontrivial domain model. If you're still not convinced, let's consider an example from our domain and discuss how you can handle this side effect. Let's take the example use case of modeling the process of opening a checking account for a customer. One of the steps in opening the account is to verify the identity of the customer and do necessary background checks. In this operation, you have to interact with external systems (outside your banking model) for getting a properly checked background verification. The following listing is our first attempt toward modeling this behavior.

Listing 1.9 Side effects in functions

```
trait AccountService {
    def openCheckingAccount(customer: Customer, effectiveDate: Date) = {
        // does an identity verification and throws exception if not passed
        Verifications.verifyRecord(customer)
        ...
    }
}
```



Identity verification; throws exception if not passed

Account-opening logic omitted

```

        Account(accountNo, openingDate, customer.name, customer.address, ...)
    }
//... other service methods
}

```

The first thing `openCheckingAccount` does is call `verifyRecord` to verify the identity of the customer. This is a call that interacts with the external world, and maybe makes a web service call through an external gateway to do the verification. This call can fail, though, and your code needs to handle exceptions related to failing external systems. And this has nothing to do with the core domain logic of giving the customer a valid, newly opened checking account!

There's no way to avoid this situation, and you'll see many such cases as you explore the other use cases of our domain model implementation. The best way to handle this issue is to decouple the side effects as far as possible from the pure domain logic. But before looking at ways to decouple, let's once again revisit the downsides of mixing side effects and pure domain logic within the same function. Table 1.3 lists them all.

Table 1.3 Why mixing domain logic and side effects is bad

| Mixing side effects and domain logic | Why it's bad |
|--|--|
| Entanglement of domain logic and side effects. | Violates separation of concerns. Domain logic and side effects are orthogonal to each other—entanglement violates a basic software engineering principle. |
| Difficult to unit test. | If domain logic is entangled with side effects, unit testing becomes difficult. You need to resort to mocking that leads to other complications in your source code. |
| Difficult to reason about the domain logic. | You can't reason about the domain logic that's entangled with the side effect. |
| Side effects don't compose and hinder modularity of your code. | Your entangled code remains an island that can't be composed with other functions. |

You need to refactor to avoid this entanglement, and you need to ensure that side effects are decoupled from the domain logic. The next example splits the two operations into separate functions and makes sure the domain logic remains a pure function that you can unit test in isolation. That's exactly what listing 1.10 does. It introduces a new function that does the verification through interaction with the external system. After it completes, it passes on a successful `Customer` instance to the `openCheckingAccount` function that does the pure logic of opening the account.

Listing 1.10 Decoupling side effects

```

trait AccountService {
  def verifyCustomer(customer: Customer): Option[Customer] = {
    if (Verifications.verifyRecord(customer)) Some(customer)
    else None
  }
}

```

```

def openCheckingAccount(customer: Customer, effectiveDate: Date) = {
    //...
    ←———— Account-opening logic
    Account(accountNo, openingDate, customer.name, customer.address, ...)
}
object AccountService extends AccountService
{
}

```

Managing side effects is an area that can make a big difference between compositional domain models and noncompositional ones. A model that's not compositional often suffers from the baggage of boilerplates and glue code. It becomes a maintenance nightmare—difficult to manage and extend. But after you've isolated side effects from pure logic, you can write code that offers better compositionality. In our example, `openCheckingAccount` is once again pure logic. The following listing shows how to glue together `verifyCustomer` and `openCheckingAccount` and handle failures that may occur in the portion of the code that handles the side effects.

Listing 1.11 Composing identity verification and opening of an account

```

import AccountService._

val cust = getCustomer(...)
verifyCustomer(cust).map(c => openCheckingAccount(c, date))
    .getOrElse(throw new Exception(
        "Verification failed for customer"))

```



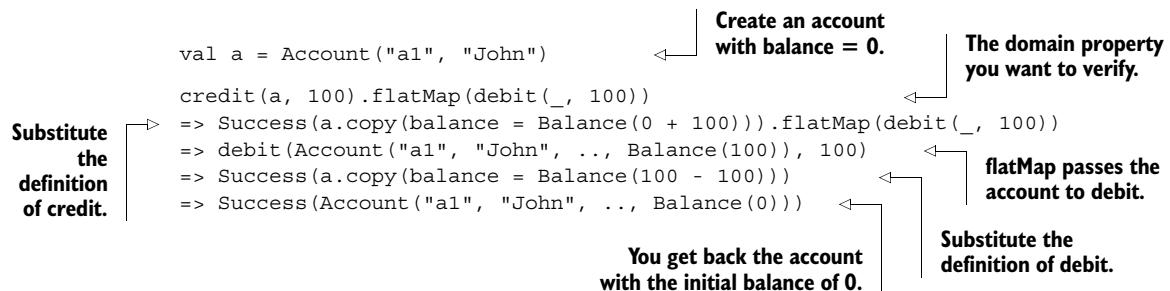
QUIZ MASTER'S RESPONSE 1.6 You already saw examples of side effects in listing 1.4, where the domain behaviors `debit` and `credit` needed to update a shared mutable state. Note that side effects are anything that depends on external systems, be it dealing with the filesystem or managing a global mutable state.

1.5 *Virtues of pure model elements*

If pure functions compose, your domain model (or at least the part of the model that has pure functions) should exhibit some mathematical properties, right? At least that was the idea when I tried to encourage modeling using pure functions. In this section, you'll try to figure out whether you can demonstrate some properties of our model and verify the correctness of it just as you can in mathematics. This is called *equational reasoning*, and you'll see many examples in later chapters.

Consider the definitions from listing 1.6, which implements `debit` and `credit` as pure functions. Using the model, you can write assertions that validate your implementation. These are called *properties* of the model, and the following code snippet provides the *verification of correctness* of the model. You start with an expression that performs a credit and a debit of equal amounts in succession to an account. This should give you the original balance of the account before you executed this expression. To validate this property, you substitute our implementation in each step of the

derivation (just as you do when you solve a mathematical equation) and finally arrive at your result.



What did you achieve in this derivation? You proved an obvious lemma that a credit of an amount x to an account following a debit of an equal amount doesn't change the initial balance of the account. Pretty obvious, isn't it?

QUIZ TIME 1.7 True or false: Equational reasoning and side effects don't go together.

Well, it looks obvious when you treat function calls similarly to how you do in mathematics. You replace one function call with its implementation and expect to achieve the same result regardless of the number of times you do this. Consider the simple example in listing 1.12.¹² In this example, you trace a function call, $f(5)$, by repeatedly replacing the invocation with the body of the function and the corresponding formal argument with the value. This almost corresponds to the way you compute functions mentally. In the theory of programming languages, this is called the *substitution model* of evaluation.

Listing 1.12 The substitution model of evaluation

```

def f(a: Int) = sum_of_squares(a + 1, a * 2)

def sum_of_squares(a: Int, b: Int) = square(a) + square(b)

def square(a: Int) = a * a

■ f(5)
■ sum_of_squares(5 + 1, 5 * 2)
■ square(6) + square(10)
■ 6 * 6 + 10 * 10
■ 36 + 100
■ 136
  
```

As you can see, by using the substitution model you get the same result every time you invoke f with an argument of 5. But be aware that the substitution model works

¹² This example is inspired by *Structure and Interpretation of Computer Programs* by Harold Abelson et al. (MIT Press, 1996).

only if functions are pure and don't have unmanaged side effects. If you had the function square also writing to a file besides returning the value, then you'd have side effects, because the file write might fail on some invocations and you'd have the function generating an exception. So the guarantee of the substitution model producing the same result on every invocation becomes void. That's another reason to favor purity of functions.



QUIZ MASTER'S RESPONSE 1.7 True. As previously discussed, side effects make computations nondeterministic, and you can't do equational reasoning with them.

Associated with the substitution model is another concept that I emphasize repeatedly in this book. Expressions such as $f(5)$ that result in the same output being generated every time you substitute the value 5 to the function's formal argument are so significant that there's a special term for them: *referentially transparent* expressions. They have a huge role to play in the world of functional programming. You can do equational reasoning only with referentially transparent expressions, as you saw earlier in this section.

So where are we on this roadmap to functional and reactive domain modeling? Here's a quick summary of the takeaways from this section:

- Referentially transparent expressions are pure.
- Referentially transparent expressions make the substitution model work.
- The substitution model helps in equational reasoning.

These three pillars of functional programming are summarized in figure 1.9.

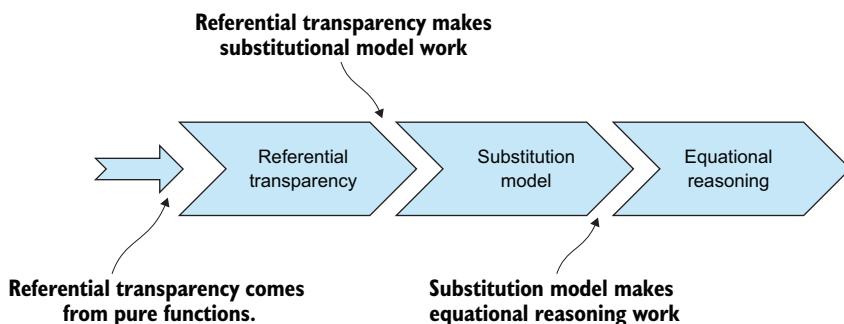


Figure 1.9 Referential transparency has a huge role to play in functional programming.

So far we've discussed the basic principles of functional programming that can help you engineer a better domain model—better in the sense that you can make your model

- More compositional by building larger functions out of smaller ones, using the power of function composition
- Pure, in that many parts of your model can be composed of referentially transparent expressions that have many of the good qualities we've discussed

- Easy to reason about and to validate many of the domain behaviors through equational reasoning

The next section focuses on an aspect that promises to make your model more responsive. Users don't like to wait for long when querying an account balance or lining up in a bank to open a checking account. Your software needs to respond to all user actions within a reasonable amount of time, often called the *latency*. And it's the property of being *reactive* that makes your model work with a bounded latency.

1.6 **Reactive domain models**

As a user, how would you feel if your request for a price quote on a newly launched, dazzling hotel-booking website took a long time to respond? Believe me, it's not always the network or the infrastructure that's to blame—the architecture of the application also has a lot to do with it. Maybe the domain model makes too many calls to underlying resources such as databases or messaging servers that have throttled the throughput of your application.

We all want our applications to respond to users' requests within an acceptable time period. This time is called the *latency*, which is more formally defined as the time period that elapses between the request that you make and the response that you get back from the server. If you can bind this latency to a limit acceptable to your users, you've achieved *responsiveness*. And being responsive is the primary criterion of your model being *reactive*. But how do you address failures? A failed system that's stuck is also not responsive. The key to addressing this issue is to design your system around failures, and you'll see more of this in section 1.6.2.

What are the primary characteristics of a reactive model? Table 1.4 summarizes the main criteria.

Table 1.4 What makes a model reactive: the four attributes

| Criterion | Explanation |
|--------------------------------|---|
| Responsive to user interaction | Otherwise, no one will use your application. |
| Resilient | This means being responsive to failures. If your system is stuck in a nondeterministic state in the face of failures, you've failed to deliver a stable model. It has to respond either by restarting parts of the application model or by giving appropriate feedback to the user regarding the next step of action. |
| Elastic | This means being responsive to varying load. Systems can face spikes of load and should be able to maintain the bounds of latency even in the face of high loads. |
| Message-driven | To stay responsive and elastic, systems must be loosely coupled and minimize blocking by using asynchronous message passing. |

1.6.1 The 3+1 view of the reactive model

If you look carefully at the four attributes in table 1.4, you'll notice that the key that makes a model reactive is it being *responsive*. The other three attributes are just characterizations of the various forms of responsiveness. This is indeed another way of looking at a reactive domain model—called the *3+1 view* of the model. Figure 1.10 illustrates this view.

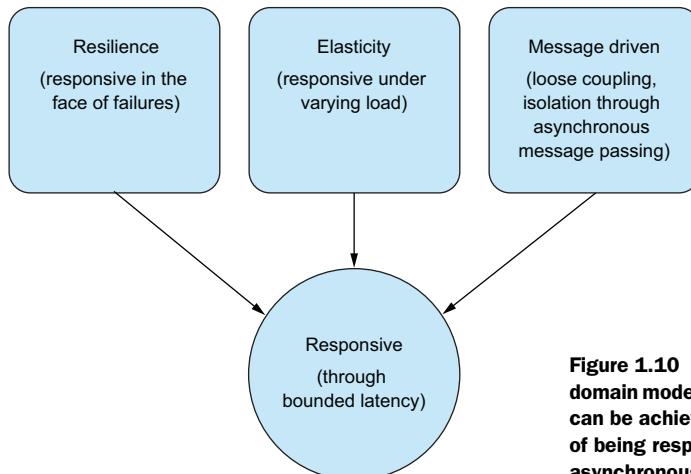


Figure 1.10 The 3+1 view of the reactive domain model: responsiveness of the model can be achieved through the various forms of being responsive to failures, load, and asynchronous messaging.



QUIZ TIME 1.8 Your favorite online bookstore works perfectly most of the year. But its response time decreases dramatically during Christmas and other holiday seasons. Which criterion of reactivity does it violate?

The model becomes reactive based on the three factors of resilience, scalability, and parallelism. Each factor has various implementation strategies, which are discussed later in this book.

1.6.2 Debunking the “My model can’t fail” myth

One of the common misconceptions that I see among naive modelers is that their model can't fail—they think they've handled every possible exception that can arise. The reality is that regardless of how robustly you think you've managed exceptions within your model, failures will occur. And the larger the scale of the model, the greater the chance for failures of components. Disks fail, memory fails, network components fail, other infrastructure fails—in short, failures can occur that are completely beyond your control.

Design for failure. This is a core concept when developing large services that are comprised of many cooperating components. Those components will fail and they will fail frequently. The components don't always

cooperate and fail independently either. Once the service has scaled beyond 10,000 servers and 50,000 disks, failures will occur multiple times a day.

—James Hamilton

“On Designing and Deploying Internet-Scale Services,”
www.usenix.org/legacy/events/lisa07/tech/full_papers/hamilton/hamilton_html/

Indeed. And one of the major teachings of the reactive model is to design *around* failures and increase the overall resiliency of the model. Table 1.4 mentions being responsive to failures; this can be achieved only if your model has the capability to handle failures not only from *within* your application, but also from other *external* sources. This doesn’t mean that you pollute your domain model with reams of exception-handling logic. The basic idea is to accept that failures are certain to occur and implement strategies to handle them explicitly as they occur in various parts of your system.

Consider an example: Say a user has requested the computation of her portfolio of various accounts with the bank. During computation, one of the steps fails, possibly because the server that holds the balance for one specific account is unreachable. How will you handle this failure in your application? There are at least a couple of approaches:

- Try including the exception-handling logic within the application code that computes the portfolio. But this can happen for any of the APIs that result in accessing back-end servers! And imagine duplicating the same code for exception handling in all such places. The result is a software engineering disaster—we call this a failure to handle *separation of concerns*. We’re conflating domain logic with exception-handling code and end up with the latter polluting the former. Clearly this doesn’t scale.
- Have a separate module that handles failures. All failures are delegated to this module, which takes responsibility for handling them based on user-defined strategies. This approach keeps your domain model clean and decouples failure handling from business logic.

Figure 1.11 summarizes the two approaches.

As you must be wondering, if all failures are handled through one module, will that be a scalability bottleneck for the entire model? How can you ensure that failure handling is as scalable as the other modules that handle the rest of your domain logic?



QUIZ MASTER’S RESPONSE 1.8 I’m sure you got it right. It’s the inability of the site to remain responsive with increasing load—which means it doesn’t scale with increased traffic. Therefore, it violates the scalability criterion.

The solution lies in the design of the failure-handling module itself. It’s not that you must have *one* single module for failure handling in the whole of your application. The idea is to have centralized handlers. The number of handlers will depend on the

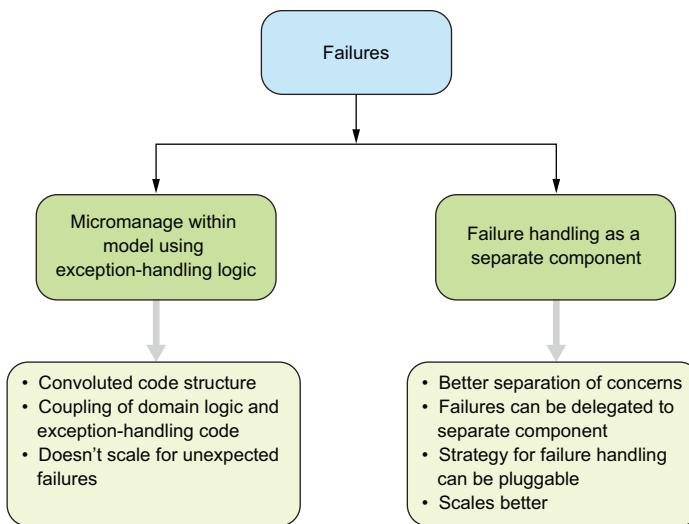


Figure 1.11 Strategies for handling failures in a domain model.
Designing a separate module for handling failures is (almost) always a better option.

overall modularity of your application. In the previous example, you may have a module that handles all failures for portfolio computation.

1.6.3 Being elastic and message driven

When we talk about elasticity, we mean the system has to be adaptive to varying load. It can be either way—you scale out when the load increases, and you scale back when it decreases. It's important to be able to scale back on decreasing load during quiet periods to ensure economy of resources and operations.

When the load on your system increases, such as during the holiday season, you see sudden spikes of latency that can violate the service-level agreement (SLA) constraints of your client. Being elastic implies that your system should be able to scale to adjust to the varying degrees of latency.

One of the ways you can make your system elastic is by reducing the coupling between the components of your model. Loosely connected architectures that use asynchronous message boundaries as the means of communication is one way to get there. That's exactly what reactive models encourage—nonblocking communication, and components that interact using immutable messages without any sharing of mutable state. When your components interact with asynchronous messages, you have the proper level of isolation because you can afford to have transparency of location, concurrency models, and the programming language itself.

This book focuses on messaging systems using the actor model of computation. Actors provide a fairly high level of concurrency construct that helps you organize

your model in terms of loosely coupled modules (they can be bounded contexts as well) that interact using asynchronous messages. We'll discuss how a well-engineered actor system can provide resilience to your model, help you scale out and back through proper handling of back pressure, and provide overall responsiveness to your system.

When I say *message driven*, I'm intentionally being a bit generic. Events can also be thought of as messages that encapsulate a domain concept. When I say a *debit* message, it's an event that induces a debit operation on specific accounts. And *debit* is a domain concept. The next section covers how domain events can form one of the first class constructs of composing behaviors within your reactive model.

1.7 Event-driven programming

With just a little idea of what events are, let's start with an example from our domain of personal banking. You'll begin with a model in which all function calls are synchronous, and blocking and execution are completely sequential. You'll study the drawbacks of this model and try to improve on the responsiveness by introducing an event-driven architecture.

Consider the following function, where you as a client ask for a portfolio statement of all holdings from the bank. Here are some of the typical line items that need to be fetched, computed, and aggregated to generate the statement:

- General currency holdings
- Equity holdings
- Debt holdings
- Loan information
- Retirement fund valuation

A sample implementation of these items can have the structure shown in the following listing.

Listing 1.13 Portfolio statement

```
val curr: Balance = getCurrencyBalance(..)
val eq: Balance = getEquityBalance(..)
val debt: Balance = getDebtBalance(..)
val loan: Balance = getLoanInformation(..)
val retire: Balance = getRetirementFundBalance(..)
val portfolio = generatePortfolio(curr, eq, debt, loan, retire)
```

As listing 1.13 suggests, it's a sequential piece of code, all of which gets executed, blocking the main thread of execution. One function gets to execute only when the previous function in sequence has completed and has made its result available in the main thread of execution. The result is that the total latency of computation is the sum of the latencies of all individual functions, as you can see in figure 1.12.

This situation can hurt responsiveness because some of the functions may be accessing databases or other infrastructure that could take quite some time to respond. As a

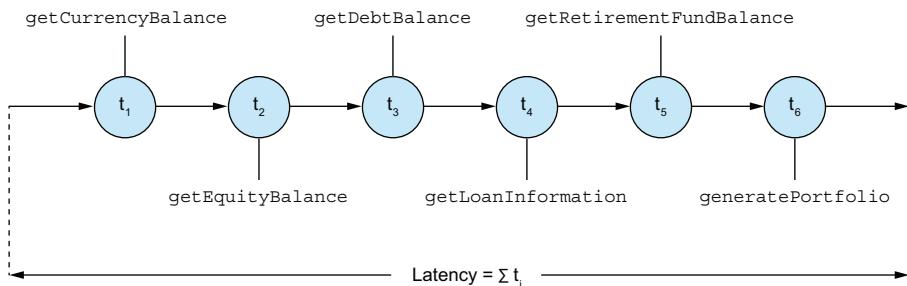


Figure 1.12 In a sequential execution, the latency of the total computation is the sum of the latencies of individual functions. There's no parallelization, and the computation is strictly sequential.

user, you wouldn't want to stare at your screen as the back-end infrastructure trudges along with a big piece of sequential computation. As you'll soon see, events offer a respite from this troubling experience.

Another way that the previous code fails to live up to its promise is that its architecture is hardwired to a local execution model.¹³ This falls apart badly when you have network communication and the various components of your portfolio need to be fetched from a cluster of computers using multiple services rather than a single machine. This is yet another area that an event-based model addresses as a solution.

Let's turn the previous code inside out and organize it in a manner that distributes the processing across multiple parallel units of computation, keeping the main thread in the role of a coordinator (see the following listing).

Listing 1.14 Portfolio statement—event driven

```

val fc curr: Future[Balance] = getCurrencyBalance...
val feq: Future[Balance] = getEquityBalance...
val fdebt: Future[Balance] = getDebtBalance...
val floan: Future[Balance] = getLoanInformation...
val fretire: Future[Balance] = getRetirementFundBalance...

val portfolio: Future[Portfolio] =
  for {
    c <- fc curr
    e <- feq
    d <- fdebt
    l <- floan
    r <- fretire
  } yield generatePortfolio(c, e, d, l, r)
  
```

¹³ When the execution model is sequential and blocking, as in this case, you can make remote procedure calls and present to the user the façade of a local execution model. But that never scales and has been found to be a victim of the fallacies of the distributed computing model. See “Fallacies of Distributed Computing Explained” by Arnon Rotem-Gal-Oz (www.rgoarchitects.com/Files/fallacies.pdf).

Here, each of the individual functions no longer makes the promise that it will return a Balance before giving control back to the main thread of execution. Instead, it returns a Future, which is sort of a placeholder for the computation. A Future makes a promise that it will give you a Balance *eventually*, when the computation of the function completes.

But as an immediate effect, it doesn't block the main thread. The main thread can continue with other tasks while the individual function goes back to complete its promise using *another thread of execution*. The net result is that all individual functions execute in their respective threads, leaving the main thread as a mere coordinator and aggregator of results. And that's exactly what happens in the call to generatePortfolio—it collects all results when they arrive and then computes the portfolio statement.

If this is an expensive operation, you can also delegate this computation to a Future (as in listing 1.14). As an astute reader, you must have already realized that in this scenario the total latency of computation is the *maximum* of all individual latencies involved in executing the individual balance computing functions (as opposed to *sum* in the sequential case), plus the latency of computing generatePortfolio.¹⁴ But because you also delegate the overall computing function to a Future, the main thread of execution becomes free to serve other requests without having to wait for completion of *any* of these functions. Note that you've already achieved an increase in responsiveness!¹⁵

Computing with Future as in listing 1.14 has transformed the sequential, blocking code of listing 1.13 into code that's asynchronous and nonblocking. When a computation returns a Future, it's equivalent to sending an event to the calling thread that says "I'll make the result available to you when I'm done." And when the computation is indeed done, the calling thread gets an event indicating the availability of the result. It can then fetch the result from the callback that it registered earlier. What you saw just now is one form of *event-driven programming*, where the events are implicitly sent to the calling thread by the Future abstraction.

1.7.1 Events and commands

I'm sure you're now comfortable with seeing events as small messages that enable nonblocking programming models. You saw how futures interacted with the main thread of execution to deliver parallelism to our domain model. When you work on completely nontrivial domain models, you'll encounter many such scenarios whereby events can make your model more responsive.

¹⁴ To fully understand why the total time is the *maximum* instead of the *sum*, you need to wait until chapter 6, where this example is discussed in detail.

¹⁵ Having talked about such performance improvement using events and asynchronous programming models, it's also not an absolute truth that all computations gain by being asynchronous and nonblocking. CPU-intensive operations can often gain by being blocking, as they can take advantage of cache coherency and lack of scheduler overhead.

Consider an event, `Debit`, that debits money from your account. Now your account balance has changed. If you maintain a balance in a database, the event triggers updates that will change the balance. And if you also maintain an in-memory copy of your aggregate, then you'll need to refresh it as well with the new balance of your account.

As soon as your account is debited, you get a message from your bank on your smart phone stating that amount x has been debited from your account in the form of a cash withdrawal. Let's name this message as well—`DebitOccurred`. In fact, this is also an event.

Do you see any difference between these two types of interactions in the model? Table 1.5 explains the differences.

Table 1.5 Two kinds of events explained. Because of differing characteristics, sometimes `Debit` is called a command and `DebitOccurred` an event.

| Debit | DebitOccurred |
|--|--|
| Has an effect on the global state of the system—equivalent to a write operation on the aggregate, in the sense that it changes the balance of an account | Is just a notification sent to interested subscribers—in this case, the holders of the account |
| Is a message that an object in the system sends <i>prior</i> to the effect taking place within the system | Is a message that's sent by the system <i>after</i> the effect has occurred |
| As a mutating message, is usually processed by a single handler of the system | Can be handled by multiple parties, each responding differently to the message |
| Can fail if some of the constraints are violated | Can't fail because the associated effect has already taken place within the system |

We call `Debit` a *command* and `DebitOccurred` an *event*. Both are messages generated and processed by the model, but they differ in semantics in a subtle way. Take note of these differences, as you'll be dealing with commands and events quite differently when we talk about our domain model architecture.

1.7.2 Domain events

Event-driven programming models make events an important architectural element. Events trigger domain logic and take part in various interactions within the domain model. In more general terms, an event is a form of notification. In the previous section, you saw two types of notifications that have slightly different semantics: commands and events. We'll frequently use the term *event* to refer to both types of notifications and make an exception only when commands need to be handled differently.

In the preceding section, you saw an example of the `Debit` event, which triggers a debit from your bank account, and a `DebitOccurred` event, which notifies interested parties that a debit has occurred from some account. You've named these events based on the action that they perform within the domain model; these events speak the language of the domain. They're known as *domain events*.

One important characteristic of events is that they're immutable. This is sort of intuitive, because you've seen that events are things that have already happened in the system. So, how can you change things that have taken place in the past?

As a customer of the bank, Bob had an address, A, on March 1, 1989. He moved to a different address, B, on June 6, 2010. One common way to deal with this situation is to update Bob's address in the customer entity. But when you're talking about events, how can you issue an update statement and change Bob's address in the customer record? The fact that he's living at address B now doesn't invalidate the fact that he used to live at address A. Some of you may say that, well, the database server maintains a log that still indicates that sometime in the past Bob lived at address A. But, after all, the database log isn't part of our model, whereas the fact that Bob lived at address A sometime back is very much part of the semantics of the domain model that we're designing. Let's look at the sequence of events happening in our domain model; table 1.6 shows the history of changes in Bob's timeline.

Table 1.6 Sequence of events in Bob's timeline

| Time | Action | Event |
|------|--|--|
| T0 | Bob opens an account with address recorded as A. | Triggers an event AddressChanged (Bob, _, A, T0) |
| T1 | Bob's address changes to B. | Triggers an event AddressChanged (Bob, A, B, T1) |

Here we're talking about a way of modeling that's different from the standard relational database-based models. We're talking about events that can't be changed, and yet they're applied on the domain model to arrive at the current snapshot. In the previous example, you can apply the last event on Bob's record in the bank to get his current address. Yet you don't lose the information that he used to be at a different address sometime back. This means that you have not only the current snapshot of the model, but also a log of the entire history that generated this snapshot. Figure 1.13 makes this distinction clear.

As mentioned earlier, domain events have an important role to play in architecting models that scale and are responsive. So far I've given you a brief introduction to what they are and how they help you build models that retain the entire history since evolution. Such domain models are called *self-tracing models*, because domain event logs make our models traceable at any point in time.

So domain events are an important participant in reactive domain models, and you must give them due importance when you build your own models. Jonas Bonér gives a nice summary of what domain events are in one of his presentations on event-driven architecture.¹⁶ He explains that domain events are

- *Uniquely identifiable as a type*—For each event, you have a type in your model.
- *Self-contained as a behavior*—Every domain event contains all information relevant to the change that just occurred in the system.

¹⁶ See “Building Loosely Coupled and Scalable Systems Using Event-Driven Architecture” by Jonas Bonér, Patrik Nordwall, and Andreas Källbärg (www.slideshare.net/jboner/event-drivenarchitecture-6097206).

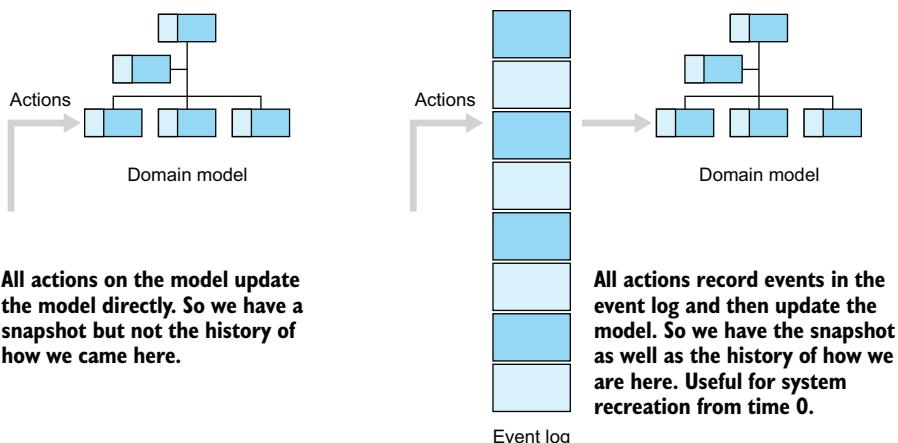


Figure 1.13 Comparing two modeling styles. The model on the left accepts all actions and applies them as updates to the current snapshot. Therefore, you lose the history of evolution. The model on the right has all events stored in a log. So you have the history as well as the ability to re-create the entire model from any point in time.

- *Observable by consumers*—Events are meant to be consumed for further action by downstream components of your model.
- *Time relevant*—Possibly the most important characteristic of a domain event. A monotonicity of time is built into the stream of events.

If the domain model can be constructed by applying all the domain events since time t_0 , then your entire model boils down to the following mathematical equation:

$$M(t_n) = \Sigma(\text{all events from time } t_0 \text{ till } t_n)$$

where $M(t_n)$ is the state of the model expressed as a sum of all events starting from time t_0 .

In later chapters, you'll see how this equation maps to an equivalent concept in functional programming.

1.8 Functional meets reactive

You've reached the end of chapter 1, which introduces the basic concepts behind implementing domain models using functional thinking and reactive processing. Why exactly are we talking about these two paradigms together? Do they make a better match than the techniques you use today in your programming? Let's look back at some of the concepts you explored in this chapter.

Any application that you implement and deliver to your client needs to be responsive. And as you saw, reactive models deliver responsiveness to failures, varying load, and concurrency. To make your model responsive to failures, you need to have resiliency

built into the architecture. And you can do this by using proper modularization of the model and ensuring that failure in one component doesn't bring down the entire system. Failures need to be handled separately instead of being coupled with the domain logic. One way to keep your model responsive even in the face of varying load is to make it event driven—the main thread of execution is never blocked. So the user requests are never stalled even though some of the requests may be doing some heavy-duty processing to serve some other request. Events are small, immutable abstractions that carry the intent of processing and get processed by an event loop. Every event that the loop gets is forked off to an event handler that does the actual job.

The reactive model implies good modularization of your code so that various event handlers can run independently and work on executing domain behaviors. And you can run processes independently only when there's no (or minimal) sharing of state between them. Functional programming encourages this practice right from the beginning. Designing with pure functions and decoupling side effects from pure logic are the two basic tenets that functional thinking brings to the forefront. And here's where functional meets reactive. Pure, referentially transparent modules acting as event handlers can be made to run concurrently to execute domain logic, making your model responsive and scalable. Pure logic scales and side effects don't, and that's where you gain a lot when you combine functional thinking with reactive modeling.

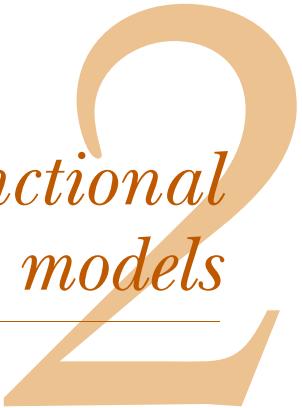
1.9 **Summary**

This chapter begins our journey into the world of domain modeling using functional and reactive paradigms. You have just learned about some of the virtues of both paradigms. Functional programming is based on function composition: You build abstractions by composing functions as first-class artifacts of the language. You use reactive principles to make your application responsive. Here are some of the major takeaways from this chapter:

- *Avoid shared mutable state within your model*—Shared mutable state is difficult to manage and leads to nondeterminism in semantics that makes concurrency difficult.
- *Referential transparency*—Functional programming gives you the power to design referentially transparent (pure) model components. When most of your model behaviors are built out of *pure* functions, you get the power of compositionality; you can build larger functions out of smaller ones through composition.
- *Organic growth*—With functional design and thinking, your model grows organically. Because it is pure, your model can be treated mathematically and you can reason about it.
- *Focus on the core domain*—When you build your model by using the principles of domain-driven design, you have entities, value objects, and services organized around patterns like repositories and factories. And you can make each of these artifacts functional. Violate the principles of purity and referential transparency as an exception, but you must be able to justify the reason for doing so. Mutability

makes some parts of your code run faster but at the same time difficult to reason about. Strive for immutability in each layer of your DDD code—this is where *functional meets DDD*.

- *Functional makes reactive easier*—Pure functions are ideal candidates for reactive modeling, because you can freely distribute them in a parallel setting without any concern for managing mutable shared state. This is where *functional meets reactive*.
- *Design for failure*—In your model, never assume that things won’t fail. Always design for failure and manage failures as a separate concern without coupling exception handlers with business logic code.
- *Event-based modeling complements the functional model*—Event-based programming delineates the “what” from the “how” of your model. And this is also what functional programming encourages. Events are small messages that specify what you want to do, and the handler for the event describes the “how” part. No wonder functional programming and event-driven programming play well together.



Scala for functional domain models

This chapter covers

- Understanding why Scala is one of the better languages for domain modeling
- Understanding the benefits of domain modeling using a statically typed language
- Combining the OO and FP power of Scala to achieve modular and pure models

Now that you're familiar with the basic concepts of functional and reactive domain modeling, how will you go about implementing such a model? And when you think about implementation, you must think about which language to use. Many languages provide enough power to implement expressive domain models. This book uses Scala, an object-functional language that's statically typed, runs on the Java virtual machine (JVM), and offers excellent interoperability with Java.

But you may wonder, why Scala? What special features does Scala offer that make it an attractive proposition for domain model implementation? You'll learn some of those features in this chapter and see how they help in specific aspects of building the model. Scala has both object-oriented (OO) and functional capabilities, which turn out to be a powerful combination in implementing and organizing domain models. You'll use the functional power of Scala to implement immutable

data and domain behaviors, and you'll use its OO capabilities to modularize your domain models.

For a more illustrative summary of how you'll progress through this chapter's sections and how you'll increase your knowledge of building functional and reactive domain models, figure 2.1 shows a sketch.

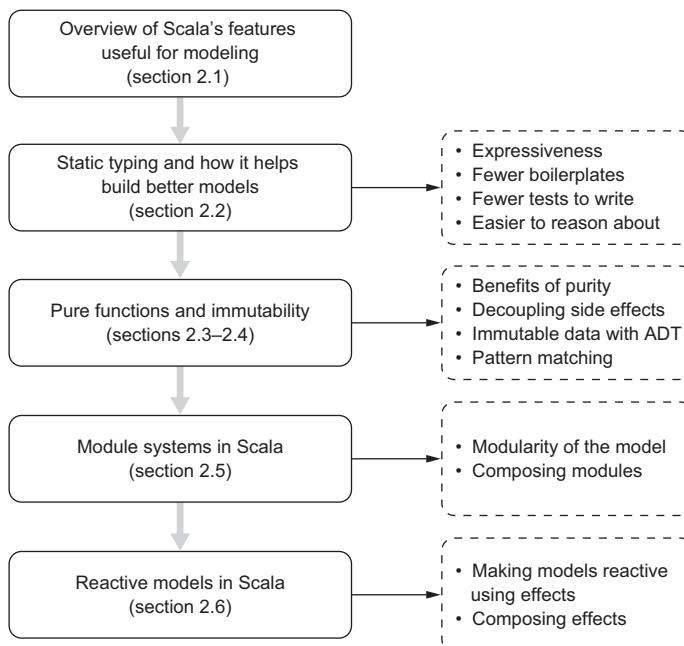


Figure 2.1 The progression of this chapter's sections

2.1 Why Scala?

Before you look into the specific features that make Scala a suitable language for implementing functional and reactive domain models, let's briefly recap what you learned in chapter 1 about some of the desirable characteristics your model should have. The two principal traits that describe our modeling paradigm are *functional* and *reactive*. When I say a model is *functional*, I mean that the model behaviors are implemented as functions with side effects clearly delineated from pure business logic. As you've seen, this helps the evolution of the model in a piecemeal way through function composition and enables you to reason about your model by using equational reasoning. When I talk about a model being *reactive*, I mean responsiveness in various forms—responsive to failure and responsive to varying load—and an overall architecture of the model that ensures the user is never a victim of unbounded latency.

To build a system that's well engineered and at the same time responsive to your users, you need to make the right architectural decisions up front. It may not be the most important architectural decision, but choosing the implementation language

with the correct set of features is one of the primary factors that can impact the overall stability of your system.

You should consider many factors before deciding on the language you want to use. You could choose a language that is dynamically typed, provides a rapid application development platform, and has a vibrant community. You can get your first version of the model out the door fast. But your model may not age well—as you go on adding more and more features, you may find yourself wishing that your language offered better support for modularity and encapsulation.

Another point to consider is performance. Statically typed languages offer better performance than dynamically typed ones, mainly because with added type information the compiler can do certain optimizations. But you may still decide to go for a dynamically typed language, thinking that you can address the performance issue by adding more servers to your deployment stack. Social issues may be involved as well—your team may have expertise in certain languages, and this can dictate the choice of your implementation language.

This book uses Scala as the implementation language. Some of the macro-level features of Scala have made it hugely popular for server-side development. In this chapter, you'll see how Scala's features map to our requirements in developing a functional and reactive domain model. Note that I'm not doing an exhaustive Scala tutorial here—my goal is to reveal how Scala helps you develop a better model through some of its built-in features (see table 2.1).

This section provides a bird's-eye view of this mapping. Subsequent sections elaborate with examples. If you're experienced with Scala, you may want to take a look at the current section and skip the rest of the chapter.

Table 2.1 A bird's-eye view of how major features in Scala map to easier development of functional and reactive domain elements

| Scala feature | Model concepts |
|--|---|
| Algebraic data types (case classes) with built-in support for immutability | Help modeling domain objects—entities and value objects; for example, Bank, Account, and so forth as domain entities. |
| Pure functions | Help model domain behavior; for example, implementing the logic of debit, credit, and so forth in a personal banking system. |
| Function composition and higher-order functions | Compose smaller behaviors to implement larger ones; for example, you can compose debit and credit to implement the logic of transferring funds between two accounts. |
| Advanced static type system with type inference | Helps make your model more robust by encapsulating some of the constraints and business logic within the type itself. Type inference helps make code concise because the compiler can infer types from the expression. |
| Traits and objects that compose | Scala traits help in modularization. You can organize your model as objects composed of multiple traits that implement various functionalities. The traits can also be parameterized with types that allow you to plug in behaviors corresponding to specific business rules. |

Table 2.1 A bird's-eye view of how major features in Scala map to easier development of functional and reactive domain elements

| Scala feature | Model concepts |
|---|--|
| Support for generics | Helps you build abstractions on generic types that can be later instantiated for specific ones. For example, you can define a domain service, <code>PortfolioService[C]</code> , for a generic customer type, <code>C</code> , that models the common workflow of the service. You can then specialize the variable parts by redefining them for every type of customer in your model. |
| Support for concurrency models like the actor model of computation that builds on top of the Java concurrency model | Scala supports abstractions for concurrency like actors and futures that help you model reactive nonblocking elements without writing low-level code that uses threads and locks. |

In the following sections, you'll take a detailed look at the features in table 2.1 with examples from our personal banking system.

2.2 Static types and rich domain models

Let's consider examples from our personal banking domain that we started in chapter 1. Let's say that in a bank, a customer account is interest bearing only if it's a savings account; checking accounts don't offer any interest. How will you model this when you're implementing a function, `calculateInterest`, that accepts a customer account and computes interest for the account for a given period of time? The following listing shows the first attempt.

Listing 2.1 Calculating interest for a customer account

```
case class Account(..)
def calculateInterest(account: Account, period: DateRange) = {
  if (!(account.accountType == SAVINGS))
    Failure(new Exception(s"$account has to be a savings account"))
  Success(..)
}
```



Logic for computing interest goes here.

As per the requirement, you raise an error if the input account isn't a savings account and compute the interest otherwise. When you write unit tests for this function, you'll have test cases that check whether the interest is computed only for the specific type of account for which it's applicable. The function is supposed to raise an error if you pass it an `Account` type other than savings, and your tests should reflect this business validation as well.

Let's consider a slight variation in the implementation. Instead of passing a concrete `Account` type, let's make the function *polymorphic*. By polymorphic, I mean that you no longer pass a concrete data type for `Account` as you did with the function `calculateInterest` in listing 2.1. Instead, you parameterize the function on the type of

account that it takes. This makes the function polymorphic with respect to the account type that it can handle. You can make it an unconstrained generic type `A` that indicates that you can pass an account of *any* type to the function. In listing 2.2, you'll constrain the type parameter `A` to be a subtype of `InterestBearingAccount`. So by definition, you can't pass any account to the function that doesn't abide by this constraint. You've already encoded some of the domain logic by using the power of the type system.

Listing 2.2 Calculating interest—a polymorphic version

```
trait Account {
    def number: String
    def name: String
    //...
}
case class CheckingAccount(...) extends Account

trait InterestBearingAccount extends Account {
    def rateOfInterest: BigDecimal
}
case class SavingsAccount(...) extends InterestBearingAccount
case class MoneyMarketAccount(...) extends InterestBearingAccount

def calculateInterest[A <: InterestBearingAccount](account: A,
  ➔ period: DateRange) = {
    }

```



Logic for computing interest goes here.

Account is now a polymorphic data type, and the function `calculateInterest` is said to be polymorphic on the `Account` type. You're using Scala's type system to encode your domain logic, and encoding in the function definition the logic that interest can be calculated *only* for certain account types. Let's see how this works and what this strategy buys you in the long run:

- Domain logic is now more explicit. A separate data type encoding the account type makes the domain model more expressive.
- The function `calculateInterest` has the valid account type as part of the signature—you've constrained the generic type `A` to be a subtype of `InterestBearingAccount`. This signature will be part of the documentation. So the user of your API won't have to look at the implementation to check the valid account types that the function allows.
- By passing the information that the function can take only accounts that are subtypes of `InterestBearingAccount`, you ensure that the compiler now has more information at its disposal. It can make optimizations based on such information.
- You no longer need to write tests that validate the proper account type passed to the function. The compiler does this job for you. You'll never be able to invoke the function with an account of type anything other than `InterestBearingAccount` or one of its subtypes.

This is only an example of how you can make your domain model richer and more succinct by adding the power of types. You gain expressivity, cut out the extra fat by delegating more tests to the compiler, and get rid of boilerplates in domain logic validation. In this book, you'll use more and more typed domain models and explore all the benefits that this approach brings to your implementation.

A note about type systems

Section 2.2 just scratched the surface of how a powerful type system enforces invariants and constraints on your domain model so that you get more help from your compiler. In the example for `calculateInterest` in listing 2.1, you passed a concrete data type for `Account` and handled all logic for the various types of accounts within the same function. When you used the power of the type system in listing 2.2, you constrained the data type of `Account` so that the function body contains only the logic for computing interest for an `InterestBearingAccount`. The compiler now has the added information that this function is meant to be invoked only for interest-bearing accounts; it can discard invocation with any other types of accounts. This means the compiler search space is now better structured.

2.3 Pure functions for domain behavior

In chapter 1, you saw the virtues of modeling domain behaviors by using pure functions. Functions compose, and you can build larger abstractions out of smaller ones by using function composition. This helps you evolve your domain model organically through small and reusable components. In Scala, you can write pure functions, and you saw examples of them in chapter 1.

Let's continue exploring functions by taking our `calculateInterest` as the starting example. But let's first prepare some basic abstractions that you'll use in our domain model (see listing 2.3), including the prototype for the `calculateInterest` function. Note that the function `calculateInterest` returns a `Try[BigDecimal]` to take care of cases where interest calculation may fail. If you need to refresh what `Try` does in Scala, take a look at the sidebar in chapter 1 (just before section 1.3.2).

Listing 2.3 Base abstractions and a sample `calculateInterest` function

```
trait Account {
    def number: String
    def name: String
}

case class CheckingAccount(...) extends Account

trait InterestBearingAccount extends Account {
    def rateOfInterest: BigDecimal
}

case class SavingsAccount(...) extends InterestBearingAccount
case class MoneyMarketAccount(...) extends InterestBearingAccount
```

Base abstraction for domain entity Account

May contain other attributes

Separate types for checking and savings modeled as algebraic data types

```
trait AccountService {
  def calculateInterest[A <: InterestBearingAccount](account: A,
    period: DateRange): Try[BigDecimal] = {
    }
}
```



Next you'll see how to use composition of functions with `calculateInterest` to build larger functionality within our system (see listing 2.4). The snippets are mostly independent—the main purpose is to show you various flavors of function composition that enable you to build domain behaviors in a completely pure and referentially transparent way.

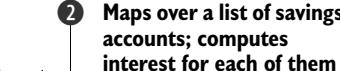
Listing 2.4 Composition that leads to evolution of abstractions

```
val s1 = SavingsAccount("dg", "sb001", 0.5)
val s2 = SavingsAccount("sr", "sb002", 0.75)
val s3 = SavingsAccount("ty", "sb003", 0.27)
```

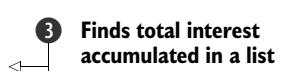


```
val dateRange = ...

List(s1, s2, s3).map(calculateInterest(_, dateRange))
```



```
List(s1, s2, s3).map(calculateInterest(_, dateRange))
  .foldLeft(BigDecimal(0))((a, e) => e.map(_ + a).getOrElse(a))
```



```
List(s1, s2, s3).map(calculateInterest(_, dateRange))
  .filter(_.isSuccess)
```

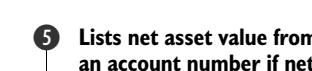


```
def getCurrencyBalance(a: Account): Try[Amount] = ...

def getAccountFrom(no: String): Try[Account] = ...

def calculateNetAssetValue(a: Account, balance: Amount): Try[Amount] = ...

val result: Try[(Account, Amount)] = for {
  s <- getAccountFrom("a1")
  b <- getCurrencyBalance(s)
  v <- calculateNetAssetValue(s, b)
  if (v > 100000)
} yield (s, v)
```



Again, the examples are simple compared to what you'd have in an actual domain model. But this simplicity helps clarify the explanation in our case. Let's look at some of the flavors of function composition that these examples demonstrate. Ideally, you can try these examples in the REPL and see for yourself how these combinators work. Each returns specific values that you can again combine for building larger abstractions.

In the first example in listing 2.3 ②, you use the `map` combinator to operate over a list of accounts defined in ①. By using `map`, you have to specify only *what* to do to each element of the list and not *how* to iterate over the list. Do you see how this is different

from the `for` loop of imperative programming? In `for` loops, you have to specify explicitly how to iterate over the list and what to do with each element in the iteration. With functional combinators such as `map`, you just need to specify the *what* part. The *how* part is abstracted within the combinator. This reduces the cognitive load on the part of the user and makes the API more expressive.

The next example ③ shows the `foldLeft` combinator. Here you're using `foldLeft`. It helps you iterate over a list of accounts and compute the sum of interest they've accumulated. Again, note that it's a pure expression and does the evaluation by accumulating the sum in every step of the iteration.

The next example ④ demonstrates the use of another combinator: `filter`. The expression obtains the list of interest calculated, excluding any interest for which the computation failed.

If you're a beginner in functional programming, I advise you to stop here for a while and try implementing the same functionalities by using an imperative language such as Java or C#. As you'll notice, in these languages you'll implement this logic of computing the sum of all interest as a *sequence* of statements, which execute one after another. Possibly you'll use `for` to loop over all the accounts and invoke the function separately on each of them and, in the process, accumulate the interest in a mutable state. The first difference that you'll notice here is that when you're using pure functions, the logic is represented as a single expression (and not a sequence of statements). Therefore, you can pipeline values from one function to another without using an intermediate state. This is the reason the compiler can apply several optimization techniques such as *fusion*, discussed in the next section. An expression has a value, which feeds straight into another expression. Note in listing 2.4 that the `map` combinator takes input from `List(s1, s2, s3)` without any intermediate counter that you'd need if you used an imperative `for` loop.

The last example in listing 2.4 ⑤ is an even more powerful demonstration of expression-oriented programming. You build expressions in a piecemeal way to arrive at the final solution without ever asking for any intermediate state. If you're still not convinced, check out figure 2.2, which shows the transformation. Statements, on the other hand, are mostly side effects, and in an imperative language, what you're doing is generating side effects in sequence to achieve the same result. The process is error prone, because mutable states are involved—therefore, functional programming is often also called *expression-oriented programming*. Figure 2.2 summarizes the difference between statements and expressions.

The final example in listing 2.4 ⑤ also illustrates a *for-comprehension* in Scala. A for-comprehension sequences the operations in a pipeline by using the `flatMap` and `map` combinators. Did I say *sequences*? That's supposed to be an imperative language style, as you just saw. Indeed, but in this case the sequencing of operations is syntactic sugar for an expressive syntax. The sequencing operation is transformed to an equivalent single expression, just as you'd do using `flatMap` and `map`. You get a double benefit: Sequencing operations is more intuitive to the user, whereas expression-oriented

| Statement | Expression |
|---|---|
| <pre>if <condition> { <statement1> } else { <statement2> }</pre> <ul style="list-style-type: none"> • Side-effecting—each statement is an assignment. • Does not return any value—only side effects. • Difficult to compose and combine. | <pre>val result = if <condition> { <expression1> } else { <expression2> }</pre> <ul style="list-style-type: none"> • Referentially transparent if individual expressions are referentially transparent. • Returns values that can be passed in a function. Note the if-expression returns a value. • Can be composed easily, because expression oriented. • Easier to reason about. |

Figure 2.2 Difference between statements and expressions. This summary explains clearly why you should choose expressions over statements when you’re thinking of functional composition.

execution gives you all the benefits of functional programming. Figure 2.3 “de-sugars” the for-comprehension into an equivalent sequence of map and flatMap.

In fact, the for-comprehension is syntactic sugar on top of a chain of flatMaps and maps. You can look up more details of Scala for-comprehensions in *Programming in*

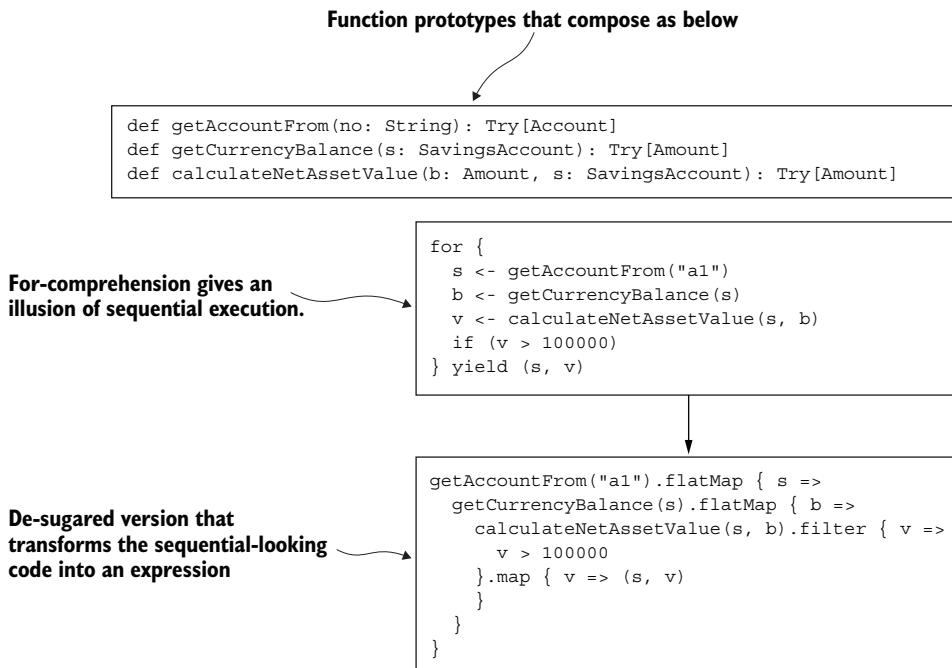


Figure 2.3 Sequencing of operations using for-comprehensions is syntactic sugar. The for-comprehension is transformed using map, flatMap, and filter. The sequence of operations is being transformed into an expression that you can compose.

Scala by Martin Odersky et al. 3rd Ed. (Artima Press, 2016). But our point of interest here is that by using such compositional structures of functional programming in Scala, you develop model behaviors that are expressive to the user. And these are robust abstractions, too.

You might be wondering what happens if any of the steps in the for-comprehension fail or raise an exception. These are all handled by the implementation of the `map` and `flatMap` combinators. If any step fails, the entire sequence is broken automatically. So error handling is yet another aspect that's taken care of by the compositionality of these abstractions.

2.3.1 Purity of abstractions, revisited

You've probably noticed that I've been using the term *pure functions* in the context of modeling domain behaviors.¹ I did this in chapter 1 as well and explained how purity helps you reason about your functions. To recap, a function is pure if it doesn't have any side effects. And what's a side effect? A side effect is something that's not within the control of the function that you implement. If you're manipulating the filesystem, or using databases or any other external resource from within your function, you are creating side effects. Doing so contradicts the definition of a pure function that's supposed to generate the same output for the same input every time you invoke it. Incidentally, pure code also has a formal name: *referentially transparent*. It means the same thing: You get the same output for the same input from an expression.

This section presents optimizations you can perform on your model with pure functions and abstractions. Let's start with an example from our personal banking domain. Consider two functions that calculate the interest on a balance in an account and deduct tax from the computed interest based on some logic. Again, the specific logic isn't important, and you'll consider some unreal and simplistic assumptions. The following listing implements these functions along with the basic abstractions.

Listing 2.5 Power of purity

```
def calculateInterest: SavingsAccount => BigDecimal = { a =>
    a.balance.amount * a.rateOfInterest
}

def deductTax: BigDecimal => BigDecimal = { interest =>
    if (interest < 1000) interest else (interest - 0.1 * interest)
}

trait Account {
    def no: String
    def name: String
    def balance: Balance
}
```

¹ You've already seen the virtues of pure abstractions in chapter 1, section 1.3.

```

case class SavingsAccount(no: String, name: String,
    balance: Balance, rate: BigDecimal) extends Account

case class Balance(amount: BigDecimal)

val a1 = SavingsAccount("a-0001", "ibm", Balance(100000), 0.12)
val a2 = SavingsAccount("a-0002", "google", Balance(2000000), 0.13)
val a3 = SavingsAccount("a-0003", "chase", Balance(125000), 0.15)

val accounts = List(a1, a2, a3)

accounts.map(calculateInterest).map(deductTax)           ← 1 Maps twice over
                                                          the collections

accounts.map(calculateInterest andThen deductTax)        ← 2
                                                          Fuses the maps by composing the functions. Note we use
                                                          “andThen” here; f andThen g means g(f(x)). There’s another
                                                          combinator, “compose”—f compose g means f(g(x)).

```

By now you’re familiar with what the code in listing 2.5 does. The technique used in the second `map` expression is called *fusion*: You fuse the two `map` combinators by using function composition. Is there any difference between the two expressions ① and ② that compute the net interest after deducting the tax?

Let’s focus on the two function definitions of `calculateInterest` and `deductTax`. Variant ①, which maps twice with `calculateInterest` and `deductTax`, behaves as shown in figure 2.4. The first `map` generates a list of interest that’s sent as input to the second `map`, which generates the list of net interest after tax deduction. So here, besides the input collection and the final output, you have an intermediate collection, which is the list of interest.

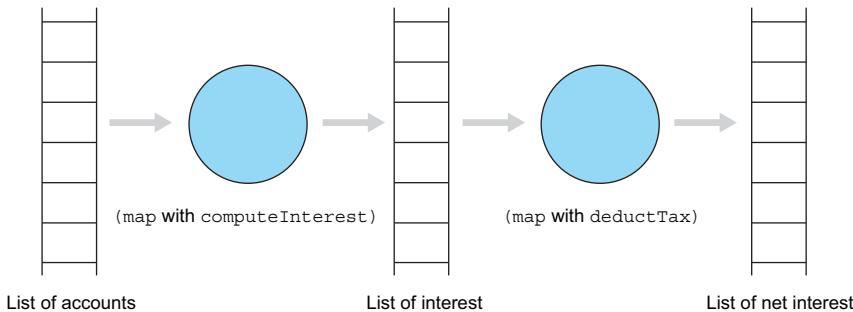


Figure 2.4 Mapping twice over a collection to generate the final list of net interest. You have an intermediate collection here, which may be a problem if the collection size is large.

In the second variant ②, you perform an optimization. Note the types of the two functions: `SavingsAccount => BigDecimal` and `BigDecimal => BigDecimal`. They line up nicely for composition. So instead of applying the two functions separately on the lists, why not apply the composition itself? After all, that’s what you’re doing in the first variant: feeding the result of the first application to the second one and

generating an intermediate collection in the process. Figure 2.5 shows the compositional approach.

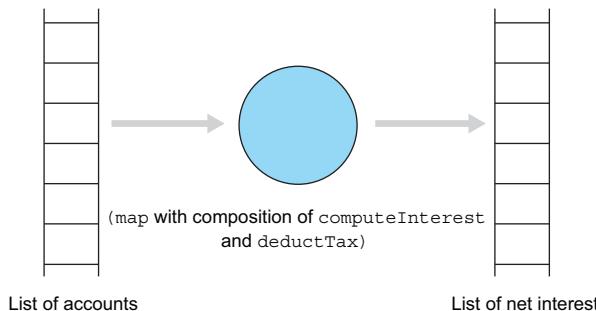


Figure 2.5 You get rid of the intermediate collection of figure 2.4 by mapping with the composed function.

Composition has given you a performance optimization. Where you find patterns of the form `list.map(f).map(g)` in your code, you can replace them with `list.map(g compose f)` and achieve this optimization. This statement assumes that you can use this optimization for `f` and `g`. But with Scala, is this a valid proposition?

Consider what happens when you choose an `f` that creates some side effects (for example, it creates a file or updates a database) and a `g` that uses the result of those side effects. This optimization trick falls flat. The functions `f` and `g` no longer depend only on the input that they get (as was the case with our `calculateInterest` and `deductTax`), but instead also depend on external resources. They're no longer referentially transparent. They have lost their purity and the ability to compose. So the net takeaway from this discussion so far is that you should strive to make functions as free from side effects as possible. Instead of having pure logic and side effects lumped together within a function, decouple them so that you achieve the benefits of compositionality at least from the pure logic portion of it.

2.3.2 Other benefits of being referentially transparent

In the previous section, you saw how to compose pure functions and perform optimization by using fusion techniques. Making your functions referentially transparent has a couple of other benefits that functional programming in general brings—nothing specific only to Scala.

TESTABILITY

Pure functions are easier to test, because you don't depend on any side effects or external state. If your function depends only on the input that you specify, the specification can be expressed as properties. You can supply these properties as expressions to property-based testing libraries such as ScalaCheck (<http://scalacheck.org>) so that they can generate random data and do the testing for you. This is a huge benefit and far more effective than the traditional xUnit-based testing. You'll learn about more benefits of property-based testing later in this book.

PARALLEL EXECUTION

If your code is side effect-free, you can use parallel data structures more effectively without any fear of external state getting in your way. In Scala, if you have a snippet of code for mapping over a collection such as `collection.map(//...)`, you can transform it for parallel execution just by changing it to `collection.par.map`. This will have the desired impact without any nasty surprises only if the function you pass to `map` is a pure one.

2.4 Algebraic data types and immutability

In this book when we discuss domain models, I'll talk about modeling entities, value objects, and other types of abstractions, and an algebraic data type (ADT) is a concept that you'll encounter. You'll need to have a clear understanding of the various types of ADTs that you can model in Scala. We all know what a data type is, but in what sense is the data type algebraic?

This section doesn't start with the theory. Instead, it begins with examples that show you what I mean by ADTs and how they can be useful when you model your domains by using them.

2.4.1 Basics: sum type and product type

In the personal banking domain, we deal with various types of currencies—USD, AUD, EUR, and INR. There are various classifications, but the base type is still a currency. The following listing shows how to model this in Scala.

Listing 2.6 Modeling currency (sum type)

```
sealed trait Currency
case object USD extends Currency
case object AUD extends Currency
case object EUR extends Currency
case object INR extends Currency
```

Here you have a base abstraction that generalizes the currency model. You also have specialized subtypes that indicate the types of currencies in your system. In the model, an instance of currency can be one of the following: USD, AUD, EUR, or INR. It can take only *one* of these values; you can't have a currency that's both a USD and an INR. So it's an OR, and in logic you represent OR by a plus: type `Currency = USD + AUD + EUR + INR`.

So you have a new data type, `Currency`. Can you figure out how many distinct values can be of type `Currency`? In terms of type theory, we call this the number of *inhabitants* of the data type `Currency`. The answer is four, which you find by *summing* the number of distinct values that the `Currency` data type can have. Yes, `Currency` is a *sum type*.

Let's take another example, this time from the Scala standard library. The following listing shows how Scala models the `Either` data type.

Listing 2.7 The Either data type in Scala

```
sealed abstract class Either[+A, +B] { //..
}
final case class Left[+A, +B](a: A) extends Either[A, B] { //..
}
final case class Right[+A, +B](b: B) extends Either[A, B] { //..
}
```

Listing 2.7 shows a base model for the `Either` data type, which takes two type parameters, and you have two specializations for `Left` and `Right`. When you construct an instance of `Either`, you can inject a value of type `A` by using the `Left` constructor, or you can inject a value of type `B` by using the `Right` constructor. So when you define an instance of an `Either`, it has to be a `Left OR a Right`, never both. This is another example of a sum type.



QUIZ TIME 2.1 For the `Either[A, B]` data type in listing 2.7, how many inhabitants can there be? Hint: It's not two.

Let's now revisit our old friend, the `Account` abstraction that we talked about earlier. The following listing shows a slightly changed version of the `Account` class in Scala.

Listing 2.8 The Account abstraction (product type)

```
sealed trait Account {
  def number: String
  def name: String
}

case class CheckingAccount(number: String, name: String,
                           dateOfOpening: Date) extends Account

case class SavingsAccount(number: String, name: String,
                           dateOfOpening: Date, rateOfInterest: BigDecimal) extends Account
```

An `Account` can be either a `CheckingAccount` or a `SavingsAccount`. This is another example of a sum type. But let's now focus on what's within a specific instance of an `Account`. A `CheckingAccount` has a `number`, a `name`, and a `dateOfOpening`. You've combined these attributes and created a new data type in order to assign new semantics to this collection of data types. In the language of types, you represent this as `(String, String, Date) => CheckingAccount` or, more generally, `type CheckingAccount = String x String x Date`. In simple terms, a `CheckingAccount` data type is the collection of all valid combinations of the tuple `(String, String, Date)`, which is nothing but the Cartesian product of these three data types. This is known as a *product type*. So in this example you have `Account` as a sum type, and each type of `Account` is a product type. Figure 2.6 describes the sum and product types and their inhabitants.

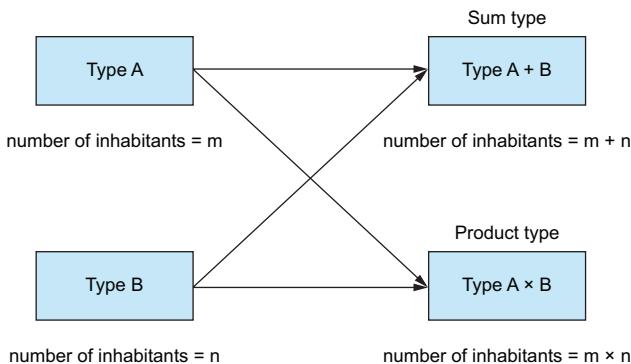


Figure 2.6 Sum types and product types in a nutshell. The number of inhabitants of each type determines the number of inhabitants in the sum and product types.



QUIZ MASTER'S RESPONSE 2.1 You can find the number of inhabitants by summing the number of inhabitants of the individual types. In the Currency example, all individual types have a single inhabitant, because each is a case object in Scala, which models a singleton. In the Either [+A, +B] example, the number of inhabitants is the number of inhabitants in A and the number of inhabitants in B. For example, if you have Either[Boolean, Unit], the number of inhabitants will be 2 (for Boolean) + 1 (for Unit), which makes it 3. I guess that was a slightly tough one!

2.4.2 ADTs structure data in the model

Sum types and product types provide the necessary abstraction for structuring the various data of our domain model. Whereas sum types let you model the variations within a particular data type, product types help cluster related data into a larger abstraction. Remember that we talked about how functional programming helps create larger abstractions from smaller ones? This is an example where proper application of ADTs makes data types compositional.

When you say `case class CheckingAccount(number: String, name: String, dateOfOpening: Date)`, you're combining a `(String, String, Date)` and tagging it with a new data type, `CheckingAccount`. Just think for a moment: You could avoid the extra tag and work with a tuple of three elements instead. But with a tagged data type, you can associate a named identity that maps to the domain that you're modeling.

Another advantage of using an ADT is that the compiler automatically validates the various tags containing valid combinations of data types. Try instantiating a `CheckingAccount` with an additional item, `rateOfInterest: BigDecimal`. The compiler will catch you immediately! Every instantiation that you make has to be a *valid* encoding from the list of enumerations that your data type has—not only does the tag have to be valid (in our case `CheckingAccount` and `SavingsAccount`), but the component data types also have to match one-to-one. The net takeaway from this section is that *an ADT forces you to build abstractions strictly according to the rules that you've defined for it*.

ADTs define the structure of data in your model. In the next section, you'll learn how to associate domain-specific behaviors with the respective tags of an ADT.

2.4.3 ADTs and pattern matching

Now that you've seen how an algebraic data type helps organize the structure of your data in the model, let's look at another related concept that works on ADTs and helps structure the functionality of our model. Pattern matching helps keep functionality local to the respective variant of the ADT. This not only adds to the readability of the model, but also makes your code more robust, as you'll see. Time to turn to an example from our domain; take a look at the following listing.

Listing 2.9 Pattern matching on ADTs

Account abstraction similar to what you implemented before

```
case class Account(no: String, name: String, dateOfOpening: Date,
  balance: Balance)
sealed trait Instrument
```

Instrument abstraction expressed as combination of sum and product types

```
case class Equity(isin: String, name: String, dateOfIssue: Date)
  extends Instrument
```

```
case class FixedIncome(isin: String, name: String, dateOfIssue: Date,
  issueCurrency: Currency, nominal: BigDecimal) extends Instrument
```

```
sealed trait Currency extends Instrument
case object USD extends Currency
case object JPY extends Currency
```

```
case class Amount(a: BigDecimal, c: Currency) {
  def +(that: Amount) = {
    require(that.c == c)
    Amount(a + that.a, c)
  }
}
```

Helper class

Balance abstraction that stores client balance for various types of Instrument

```
case class Balance(amount: BigDecimal, ins: Instrument, asOf: Date)
```

```
def getMarketValue(e: Equity, a: BigDecimal): Amount = //..
def getAccruedInterest(i: String): Amount = //..
```

```
def getHolding(account: Account): Amount = account.balance match {
  case Balance(a, c: Currency, _) => Amount(a, c)
  case Balance(a, e: Equity, _) => getMarketValue(e, a)
  case Balance(a, FixedIncome(i, _, _, c, n), _) =>
    Amount(n * a, c) + getAccruedInterest(i)
}
```

Pattern matching on a specific Instrument type (Currency and Equity)

Pattern matching and destructuring to extract values to be used in further computation

Listing 2.9 defines a couple of sum types and product types to model domain objects. I'll use these ADTs to explain how to associate domain behaviors to each of the enumerations of the ADT.

In listing 2.9, the primary function that uses pattern matching is `getHolding`, which computes the account's net holding value based on the type of `Balance` that it holds. `Instrument` is defined as a sum-of-product type, and pattern matching enables you to localize the exact logic with the sum type that you're using. Within each sum type, you have the product type, which can again be destructured with pattern matching. That enables you to pick the necessary attribute values to be used for computation. Encoding this type of implementation by using OO and subtyping will lead you to the Visitor pattern,² which, as we all know, is fraught with grave perils.³

Another big advantage of pattern matching in Scala is that the compiler checks for the exhaustiveness of the pattern match. If you forgot to insert any of the enumerations of `Balance` in the pattern match clause, the compiler will issue a warning. If you add an extra enumeration to a sum type later in time, the compiler will point to every possible place where the pattern match clauses need to be updated with the additional variant of the ADT.

2.4.4 **ADTs encourage immutability**

One essential characteristic of an algebraic data type in Scala is its encouragement of immutability. When you write `case class Person (name: String, address: String, dateOfBirth: Date)` Scala creates an immutable abstraction by default. If you need to make attributes mutable, you must have an explicit `var` with that attribute to indicate that you want mutability. But of course you don't want to go that way. Functional programming encourages immutability and eschews in-place mutation, making abstractions referentially transparent.

The question is, in Scala, once you define an ADT, how do you modify the value of any of its attributes? The quick answer is, you don't. Instead, you create another abstraction from the original one that has the modified value (see the following listing).

Listing 2.10 Scala case classes, which are immutable by default

```
case class SavingsAccount(number: String, name: String,
                           dateOfOpening: Date, rateOfInterest: BigDecimal) extends Account

val today = Calendar.getInstance.getTime
val a1 = SavingsAccount("a-123", "google", today, 0.2)           ↪ Original account
val a2 = a1.copy(rateOfInterest = 0.15)                            ↪ Copy with the
                                                               ↪ changed value
```

² Find all the gory details of Visitor and other design patterns in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, et. al (Addison-Wesley Professional, 1994).

³ The Visitor pattern in Java leads to a convoluted code base that's extremely hard to extend. This may be why there are so many variations of the pattern. For details, see “Variations on the Visitor Pattern” by Martin E. Nordberg III (<http://c2.com/cgi/wiki?VariationsOnTheVisitorPattern>). Algebraic data types solve the problem much more elegantly.

In listing 2.10, `a1` is a `SavingsAccount` that you created as an account opened today. Later you decide to change the rate of interest to 0.15 from the earlier value of 0.2. The way to do it in Scala and not forgo immutability is to use the `copy` method and create another instance of `SavingsAccount` with the changed value for `rateOfInterest`.



QUIZ TIME 2.2 Suppose you have the following ADT definition:

```
case class Address(no: Int, street: String, city: String, zip: String)
case class Identity(name: String, address: Address)
```

And you have an `Identity` created as follows:

```
val i = Identity("john", Address(12, "Monroe street", "Denver", "80231"))
```

How will you change the ZIP code to 80234 without in-place mutation? Hint: Use a nested copy.

Immutability has lot of virtues, many of which are discussed in chapter 1. It makes life so easy in a parallel and concurrent setting, where you can share values freely without any of those fears that come from mutable states.⁴ Immutable data is one of the most important principles that we practice in functional programming, and it connects well with the idea of functional purity. In your domain models, always strive to achieve as many of these ideas as possible; keep domain behaviors pure and use immutable data. In our discussion on implementing functional domain models, you'll explore higher-order abstractions such as lenses that build on top of immutable ADTs and easy-to-use combinators for manipulating them.



QUIZ MASTER'S RESPONSE 2.2 It's slightly tricky because you have to use `copy` at two levels. Here's the solution:

```
i.copy(address = i.address.copy(zip = "80234"))
```

You've seen how Scala supports some of the basic ideas of functional programming—functions as first-class values, higher-order functions, purity of functions, and immutability of data. These make Scala a good language to implement functional domain models. In the final section, you'll take a look at modularity, an aspect that makes your model evolve incrementally by organizing the whole structure into manageable pieces.

2.5

Functional in the small, OO in the large

A nontrivial domain model is bound to have many abstractions, including entities, value objects, services, and their associated behaviors. How would you feel if all these artifacts were bundled together in the same namespace and you had a monolithic

⁴ For more details on this, see *Java Concurrency in Practice* by Brian Goetz (Addison-Wesley Professional, 2006).

application at your disposal? There's a good chance that a model implemented as a monolithic structure would be a cacophony of entanglement. You wouldn't have any clear separation of responsibilities, abstractions wouldn't be properly namespaced, and you wouldn't have any bounded-context-based segregation. This would be a mess, no question.

Modules are the answer to all these problems. Various functionalities need to reside in separate modules. In the context of our personal banking system, portfolio reporting of customers can be a module, tax calculation can be another module, and auditing can be yet another module. The obvious question that you must be asking is how interconnected modules should be. After all, the entire model has to work as a whole and there need to be some interconnections even across modules. When a transaction occurs in the online customer-interaction module, that information has to flow to the auditing module. So it definitely requires some thought to make your model modular with clear separation of responsibilities.

Modules need to be loosely coupled but strongly cohesive. What does this mean? Because a module performs a definite task, it has to be cohesive within itself. The abstractions need to be small and tight, with each of them focused on doing one specific thing. On the other hand, when we talk about two different modules, the coupling between them should be as minimal as possible. It's definitely not a healthy sign to have a strong dependency between two modules—changes in one will impact the other, and this goes against the principles of modular design.

2.5.1 **Modules in Scala**

Scala offers traits and objects as implementation techniques for modular design. Using traits, you can do mixin-based composition.⁵ You can use traits to compose several smaller abstractions to build larger ones. These *aren't* functions, and I'm not talking about function composition here. Usually a single trait is a small unit of functionality containing one or a few methods focused only on delivering that functionality. Here's an example from our domain:

```
trait BalanceComputation
trait InterestCalculation
trait TaxCalculation
trait Logging

trait PortfolioGeneration extends BalanceComputation
  with InterestCalculation with TaxCalculation with Logging {
  //... implementations
}
```

As mentioned earlier, client portfolio generation is a separate module, but it consists of a few distinct functionalities, which are fairly independent. So you'll keep them as separate traits (so that they can be independently reused) but mix them together to

⁵ For more details, see *Programming in Scala*.

help build the larger functionality of portfolio generation. The important point to note is that the traits that you mix in are orthogonal to each other—for example, Logging can be reused in many other contexts, and BalanceComputation can be reused as part of customer-statement generation. Now that you have the composed functionalities necessary for portfolio generation, you can reify the module as an object:

```
object PortfolioGeneration extends PortfolioGeneration
```

One question you may ask is, do you need the trait `PortfolioGeneration`? You could've directly instantiated the object from the composing mixins. It's always a good practice to have the final module in the form of a trait before committing to a concrete implementation. Tomorrow you may need to define a bigger module that uses `PortfolioGeneration`; you can then use the intermediate trait to mix in with the other functionalities. So, it's all about modularity and composition of modules. What you saw earlier is one of the simplest techniques of modularization in Scala. Scala's type system has enough power in its arsenal that you can do sophisticated tricks and implement parameterized modules. You can leave some of the abstractions as parameters in the traits and provide only concrete instances when you create the final object. This is an extremely useful technique, and I'll show you an example from our domain that demonstrates its power.

In an implementation of the domain model, a module implements a specific business functionality. For example, tax computation of clients can be a module; portfolio generation can be another. One module is usually a part of one bounded context, though one bounded context can contain many modules.

Modules can be parameterized to incorporate variations in business rules. Say you want to compute the interest accrued in a client account for a specific period. In a sample implementation, even if you ignore the subtle business complexities, you have the following components to compute:

- 1 Calculation of interest.
- 2 Calculation of tax that needs to be deducted from the interest.
- 3 The second item needs specific tax tables containing the tax items and rate that depend on the type of transaction. (For example, you're considering interest computation here as the type of transaction, though tax can also be computed for other transaction types such as foreign exchange trade or dividend calculation.)

The subtlest part of defining modules is to consider the variability that they may have depending on business rules across deployments and implementations. In our scenario, we can say that a module for computing the net interest to be credited to the client needs to refer to another module for computation of tax to be deducted. And the module that computes the tax needs to be parameterized on tax tables that depend on the transaction type being performed (in this case, interest computation). The following listing shows the definitions of these three modules and their interdependencies.

Listing 2.11 Interest calculation modules and their dependencies

```

sealed trait TaxType
case object Tax extends TaxType
case object Fee extends TaxType
case object Commission extends TaxType

sealed trait TransactionType
case object InterestComputation extends TransactionType
case object Dividend extends TransactionType

type Amount = BigDecimal
case class Balance(amount: Amount = 0)

trait TaxCalculationTable {
    type T <: TransactionType
    val transactionType: T
}

def getTaxRates: Map[TaxType, Amount] = {
    //...
}

trait TaxCalculation {
    type S <: TaxCalculationTable
    val table: S
}

def calculate(taxOn: Amount): Amount =
    table.getTaxRates.map { case (t, r) =>
        doCompute(taxOn, r)
    }.sum

protected def doCompute(taxOn: Amount, rate: Amount): Amount = {
    taxOn * rate
}

trait SingaporeTaxCalculation extends TaxCalculation {
    def calculateGST(tax: Amount, gstRate: Amount) =
        tax * gstRate
}

trait InterestCalculation {
    type C <: TaxCalculation
    val taxCalculation: C
}

def interest(b: Balance): Option[Amount] = Some(b.amount * 0.05)

def calculate(balance: Balance): Option[Amount] =
    interest(balance).map { i =>
        i - taxCalculation.calculate(i)
    }
}

```

Tax calculation table parameterized on transaction type

Tax calculation logic parameterized on tax calculation table

Another specialization of tax calculation for a specific geography that has additional tax to be levied

Interest calculation logic parameterized on tax calculation logic

You don't have any concrete instances of the modules yet—only the module definitions and their dependencies that reflect the variabilities in our domain model. Here you're implementing variabilities by using *abstract types* and *vals* that Scala offers, which make the dependency graph explicit and expressive.

The final step in module composition is to create an instance of the module that delivers the functionality of one specific use case. In listing 2.12, you'll construct a concrete module, `InterestCalculation`, that calculates interest for non-Singapore clients by using `InterestComputation` as the transaction type. All you need to do here is specify values for all abstract types and vals to concretize all module definitions.

Listing 2.12 A concrete module for interest computation

```
object InterestTaxCalculationTable extends TaxCalculationTable {
  type T = TransactionType
  val transactionType = InterestComputation
}

object TaxCalculation extends TaxCalculation {
  type S = TaxCalculationTable
  val table = InterestTaxCalculationTable
}

object InterestCalculation extends InterestCalculation {
  type C = TaxCalculation
  val taxCalculation = TaxCalculation
}
```

Note the type and value members that were abstract in the traits `TaxCalculationTable` and `TaxCalculation` have been made concrete in the respective object declarations. The core takeaway from this strategy is that you don't need to commit to the exact type or value until you declare the object. This is awesome for module composition.

Figure 2.7 illustrates how to generate a concrete instance of a module in Scala through composition of multiple parameterized modules. The main takeaway is to keep the design flexible so that inter-module communication is minimal, yet flexible and externally injectable. The combination of objects and traits in Scala does this nicely.

You've seen a broad overview of how Scala maps to our domain-modeling techniques. The three main Scala features that make this modeling exercise a fruitful and effective one are the static type system, powers of functional programming, and first-class module support. In case you didn't have a chance to go through the entire set of examples in this chapter, figure 2.8 provides a bird's-eye view. And even if you've been studious and gone through all the examples, this figure is a useful cheat sheet for your reference.

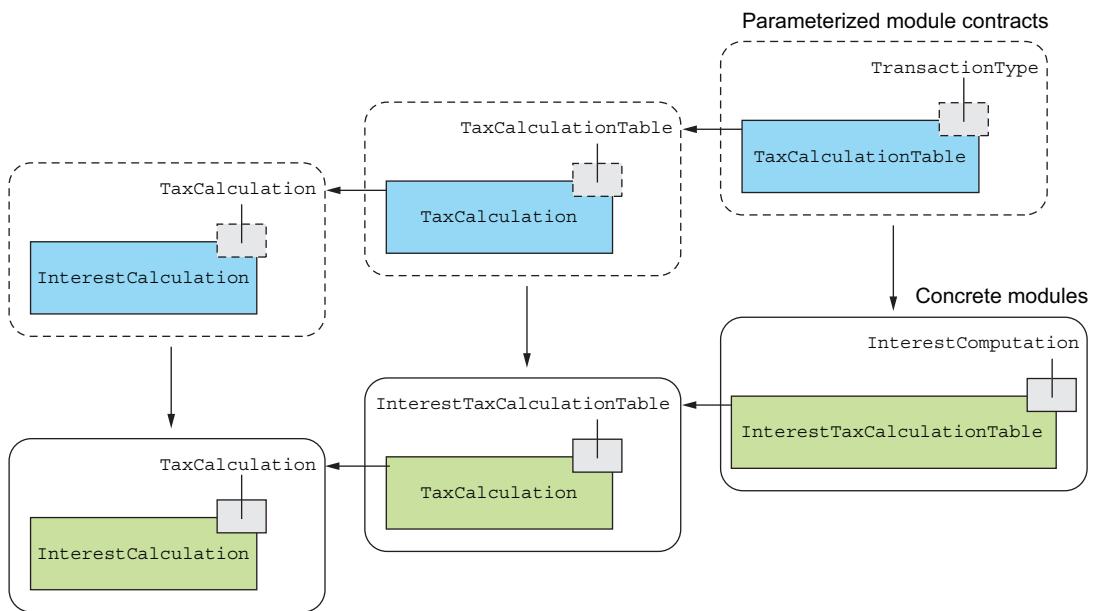


Figure 2.7 How a module evolves in Scala. You have the module contracts (shown in dotted rectangles) parameterized by types (shown as dotted squares in inset). You concretize each of them and instantiate the final concrete module.

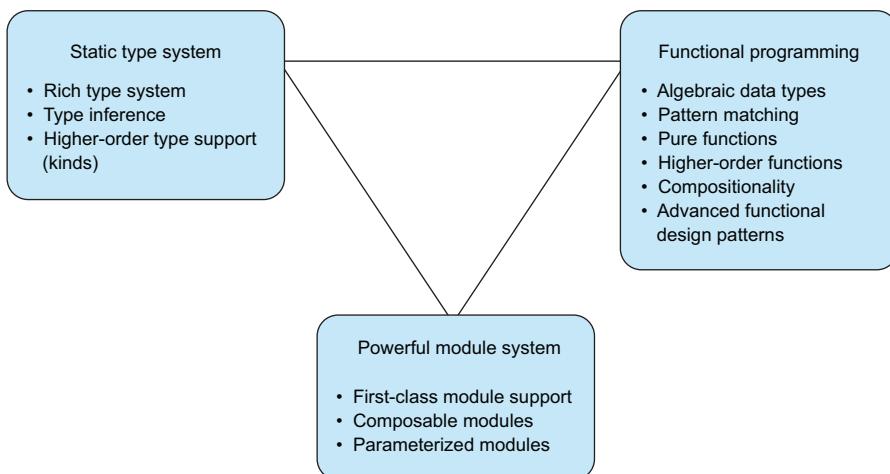


Figure 2.8 The three main features that make Scala a successful platform for domain modeling

2.6 Making models reactive with Scala

Functional thinking and implementing with pure functions is a great engineering discipline for your domain model. But you also need language support that helps build models that are responsive to failures, scales well with increasing load, and delivers a nice experience to users. Chapter 1 referred to this characteristic as *being reactive*, and identified the following two aspects that you need to address in order to make your model reactive:

- Manage failures, also known as *design for failure*
- Minimize latency by delegating long-running processes to background threads without blocking the main thread of execution

In this section, you'll see how Scala offers abstractions that help you address both of these issues. You can manage exceptions and latency as *effects* that compose along with the other pure abstractions of your domain model. An effect adds capabilities to your computation so that you don't need to use side effects to model them. The sidebar "What is an *effectful* computation?" details what I mean.

Managing exceptions is a key component of reactive models—you need to ensure that a failing component doesn't bring down the entire application. And managing latency is another key aspect that you need to take care of—unbounded latency through blocking calls in your application is a severe antipattern of good user experience. Luckily, Scala covers both of them by providing abstractions as part of the standard library.

What is an *effectful* computation?

In functional programming, an effect adds some capabilities to a computation. And because we're dealing with a statically typed language, these capabilities come in the form of more power from the type system. An effect is modeled usually in the form of a type constructor that constructs types with these additional capabilities. Say you have any type `A` and you'd like to add the capability of aggregation, so that you can treat a collection of `A` as a separate type. You do this by constructing a type `List[A]` (for which the corresponding type constructor is `List`), which adds the effect of aggregation on `A`. Similarly, you can have `Option[A]` that adds the capability of optionality for the type `A`. In the next section, you'll learn how to use type constructors such as `Try` and `Future` to model the effects of exceptions and latency, respectively. In chapter 4 we'll discuss more advanced effect handling using applicatives and monads.

2.6.1 Managing effects

When we're talking about exceptions, latency, and so forth in the context of making your model reactive, you must be wondering how to adapt these concepts into the realm of functional programming. Chapter 1 called these *side effects* and warned you about the cast of gloom that they bring to your pure domain logic. Now that we're talking about managing them to make our models reactive, how should you treat

them as part of your model so that they can be composed in a referentially transparent way along with the other domain elements?

In Scala, you treat them as *effects*, in the sense that you abstract them within containers that expose functional interfaces to the world. The most common example of treating exceptions as effects in Scala is the Try abstraction, which you saw earlier. Try provides a *sum* type, with one of the variants (Failure) abstracting the exception that your computation can raise. Try wraps the effect of exceptions within itself and provides a purely functional interface to the user. In the more general sense of the term, Try is a monad. There are some other examples of effect handling as well. Latency is another example, which you can treat as an effect—instead of exposing the model to the vagaries of unbounded latency, you use constructs such as Future that act as abstractions to manage latency. You’ll see some examples shortly.

The concept of a monad comes from category theory. This book doesn’t go into the theoretical underpinnings of a monad as a category. It focuses more on monads as abstract computations that help you mimic the effects of typically impure actions such as exceptions, I/O, continuations, and so forth while providing a functional interface to your users. And we’ll limit ourselves to the monadic implementations that some of the Scala abstractions such as Try and Future offer. The only takeaway on what a monad does in the context of functional and reactive domain modeling is that it abstracts effects and lets you play with a pure functional interface that composes nicely with the other components.

2.6.2 **Managing failures**

Chapter 1 explored the design-for-failure paradigm. Regardless of how you design your model and how much safeguarding you include for addressing exceptions, failures are bound to happen. Hardware fails, networks fail, third-party software fails, even your own software components that you’ve carefully crafted with all conceivable defensive strategies fail. So as a model designer, what strategy should you adopt to address such an all-pervasive failure situation?

As you saw in chapter 1, littering your code with error checks isn’t a solution. It doesn’t work and has been shown to be a poor practice from a software engineering perspective. Your core domain logic gets clouded in the reams of error-checking code.

Scala provides a two-pronged strategy to handle exceptions:

- Make it explicit that a portion of your code can raise an exception. Use the type system to help.
- Use abstractions that don’t leak exception management details within your domain logic so that the core logic remains functionally compositional.

Consider a function, `def getCurrencyBalance(account: SavingsAccount): BigDecimal`. Under normal circumstances, which I call the *happy path*, the function fetches the balance as a `BigDecimal`. But what about exceptions? In some cases, the function can throw exceptions, which, by the way, isn’t documented in any part of the function signature. In Scala you can do a lot better by managing failures as effects.

Let's look at concrete examples from our domain to understand the real virtues of managing exceptions in a truly functional manner. Listing 2.13 shows three functions, each of which can fail in some circumstances. We make that fact loud and clear—instead of `BigDecimal`, these functions return `Try[BigDecimal]`. You've seen `Try` before and know how it abstracts exceptions within your Scala code. Here, by returning a `Try`, the function makes it explicit that it can fail. If all is good and happy, you get the result in the `Success` branch of the `Try`, and if there's an exception, then you get it from the `Failure` variant. The most important point to note is that the exception never escapes from the `Try` as an unwanted side effect to pollute your compositional code. Thus you've achieved the first promise of the two-pronged strategy of failure management in Scala: being explicit about the fact that this function can fail.

Listing 2.13 Managing failures in Scala

```
def calculateInterest[A <: SavingsAccount](account: A,
  balance: BigDecimal): Try[BigDecimal] = {
  if (acc.rate == 0) Failure(new Exception("Interest Rate not found"))
  else Success(BigDecimal(10000))
}

def getCurrencyBalance[A <: SavingsAccount](account: A): Try[BigDecimal] = {
  Success(BigDecimal(1000L))
}

def calculateNetAssetValue[A <: SavingsAccount](account: A,
  ccyBalance: BigDecimal, interest: BigDecimal): Try[BigDecimal] = {
  Success(ccyBalance + interest + 200)
}
```

But what about the promise of compositionality? Yes, `Try` also gives you that by being a monad and offering a lot of higher-order functions. Here's the `flatMap` method of `Try` that makes it a monad and helps you compose with other functions that may fail:

```
def flatMap[U](f: T => Try[U]): Try[U]
```

You saw earlier how `flatMap` binds together computations and helps you write nice, sequential for-comprehensions without forgoing the goodness of expression-oriented evaluation. You can get the same goodness with `Try` and compose code that may fail:

```
for {
  b <- getCurrencyBalance(s1)
  i <- calculateInterest(s1, b)
  v <- calculateNetAssetValue(s1, b, i)
} yield (s1, v)
```

This code handles all exceptions, composes nicely, and expresses the intent of the code in a clear and succinct manner. This is what you get when you have powerful language abstractions to back your domain model. `Try` is the abstraction that handles the exceptions, and `flatMap` is the secret sauce that lets you program through the happy

path. So you can now have domain model elements that can throw exceptions—just use the `Try` abstraction for managing the effects and you can make your code resilient to failures.

2.6.3 Managing latency

Just as `Try` manages exceptions using effects, another abstraction in the Scala library called `Future` helps you manage latency as an effect. What does that mean? Reactive programming suggests that our model needs to be resilient to variations in latency, which may occur because of increased load on the system or network delays or many other factors beyond the control of the implementer. To provide an acceptable user experience with respect to response time, our model needs to guarantee some bounds on the latency.

The idea is simple: Wrap your long-running computations in a `Future`. The computation will be delegated to a background thread, without blocking the main thread of execution. As a result, the user experience won't suffer, and you can make the result of the computation available to the user whenever you have it. Note that this result can also be a failure, in case the computation failed—so `Future` handles both latency and exceptions as effects.

`Future` is also a monad, just like `Try`, and has the `flatMap` method that helps you bind your domain logic to the happy path of computation. You've already seen this in chapter 1, listing 1.14, where you were able to bind the latency to the maximum of the slowest computation instead of the sum of each individual computation. Even in the face of failures, you can specify time-outs in future-based computations that would help bind the latency as per the SLA of the client.

Continuing our current train of thought, imagine that the functions you wrote in listing 2.13 involve network calls, and thus there's always potential for long latency associated with each of them. As I suggested earlier, let's make this explicit to the user of our API and make each of the functions return `Future` (see the following listing).

Listing 2.14 Managing latency as effects in Scala

```
def calculateInterest[A <: SavingsAccount](account: A,
  balance: BigDecimal): Future[BigDecimal] = Future {
  if (acc.rate == 0) throw new Exception("Interest Rate not found")
  else BigDecimal(10000)
}

def getCurrencyBalance[A <: SavingsAccount](account: A)
  : Future[BigDecimal] = Future {
  BigDecimal(1000L)
}

def calculateNetAssetValue[A <: SavingsAccount](account: A,
  ccyBalance: BigDecimal, interest: BigDecimal): Future[BigDecimal] =
  Future {
  ccyBalance + interest + 200
}
```

Explicit Future as return type

By using `flatMap`, you can now compose the functions sequentially to yield another Future. The net effect is that the entire computation is delegated to a background thread, and the main thread of execution remains free. Better user experience is guaranteed, and you've implemented what the reactive principles talk about—systems being resilient to variations in network latency. The following listing demonstrates the sequential composition of futures in Scala.

Listing 2.15 Sequential composition of futures

```
val result = for {
    b <- getCurrencyBalance(s4)
    i <- calculateInterest(s4, b)
    v <- calculateNetAssetValue(s4, b, i)
} yield (v)

result onComplete {
    case Success(v) => //... success
    case Failure(ex) => //... failure
}
```

Success path of the future completion

Failure path of the future completion

Here, `result` is also a Future, and you can plug in callbacks for the success and failure paths of the completed Future. If the Future completes successfully, you have the net asset value that you can pass on to the client. If it fails, you can get that exception as well and implement custom processing of the exception. Besides sequential composition using `flatMap`, Future in Scala offers many concurrent combinators where you can build multiple futures and then let them execute concurrently, resulting in asynchronous, nonblocking parallel code. We discuss implementing parallel, nonblocking code with Scala futures in chapter 6.

2.7 Summary

In this chapter, you learned about some of the core features that make Scala an appropriate functional programming language for domain modeling. You can do domain modeling in any language that you want. But some languages offer more language and library support that enable you to program at a higher level of abstraction. And Scala is one such language. The major takeaways from this chapter are as follows:

- *Powerful type system of Scala*—Scala has a powerful type system, which can be used effectively to encode some of the domain logic. Used effectively, the type system can help reduce boilerplates and unnecessary test code.
- *First-class language support for functional programming*—Scala supports functional programming as a first-class paradigm. It has support for higher-order functions and a rich set of combinators in the Scala standard library. Using functions as first-class abstractions, you can implement domain behaviors, which can be referentially transparent and thus compositional.
- *Algebraic data types and pattern-matching support*—Scala supports implementation of the ADT for modeling immutable data. The data can be collocated with the

manipulation logic by using pattern matching. In fact, the combination of ADTs and pattern matching provides a powerful means of expressing domain logic in a succinct way.

- *First-class modules*—Modularization is a key aspect of good software engineering practices, and Scala offers great support for modules. Traits and objects in Scala help define modules that can be composed and help evolve larger components out of smaller ones. So Scala is a language where the ideal paradigm of domain model implementation is functions in the small objects at large.

Designing functional domain models

This chapter covers

- Designing domain models—the functional and algebraic way
- Decoupling the algebra of the domain from its implementation
- Enforcing the laws of algebra in designing APIs
- Implementing lifecycle patterns of domain objects

Previous chapters covered the parallels between functional programming and mathematics in general, and algebra in particular. You explored algebraic data types, sum types, and product types, and you learned how to combine them to form abstractions that model your domain elements. This chapter takes this discussion to the next level; you'll begin with specifications of the model and, using algebraic composition of types, build APIs for our domain model. These APIs are contracts that obey the laws of the domain, and you'll use the algebra of types to ensure that they're correctly constructed. You'll learn how to develop APIs as algebraic specifications and write properties that verify the laws that form the business rules.

You've already seen the three basic lifecycle patterns of domain objects: aggregates, factories, and repositories. This chapter shows how to implement them in your domain model by using the basic idioms of functional programming.

Figure 3.1 illustrates how you'll progress through the various sections and increase your knowledge of using functional techniques to build domain models.

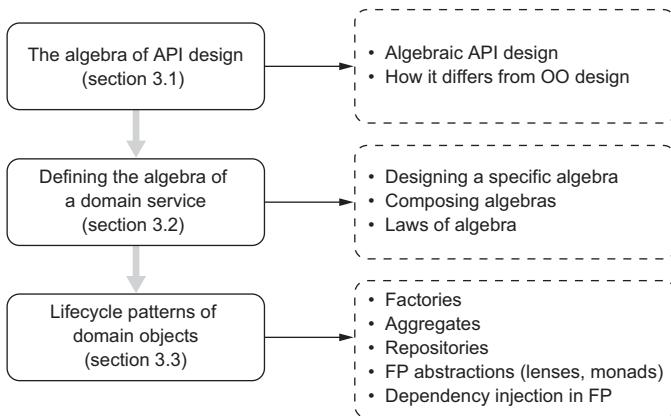


Figure 3.1 The progression of this chapter's sections

3.1 The algebra of API design

In object-oriented (OO) development using a language such as Java, you start an API design with an *interface*, which ultimately publishes the contract of the model to the end user. When you have the interface ready, you start the concrete implementation with classes and objects. You first identify the class and then group some operations as methods of that particular class. In functional programming, you turn this model inside out—you start with the *operations* that correspond to the basic domain behaviors and group related ones into *modules* that form larger use cases. Each behavior is modeled using functions that operate on types; the types represent the data, class, or object of the domain. In the next section, you'll design a domain service based on this functional paradigm.

A module, as defined in a functional domain model, is a collection of *functions* that operate on a set of *types* and honor a set of *invariants* known as domain rules. In mathematical terms, this is known as the *algebra* of the module. In case you aren't familiar with the formal definition of an algebra, I'll define it here and explain how each element maps to what we claim to be the algebra of our module:

- *One or more sets*—In our case, the sets are the data types that form part of the model.
- *One or more functions that operate on the objects of the sets*—In our case, these will be the functions that you define and publish as the API to the user.
- *A few axioms or laws that are assumed to be true and can be used to derive other theorems*—When you define operations in an API, the laws will specify the relationships between these operations.

Note that we haven't yet talked about the implementation, which is *not* part of the algebra. It's the contract the behaviors publish that forms the algebra of the API.

3.1.1 Why an algebraic approach?

You may be wondering about the advantages of this algebraic approach to API design. An API is something that has to be user focused. The first thing that a user understands about the domain is the set of behaviors, which the algebraic approach models with pure functions not clouded under the shades of any class or object. After you define the functions clearly, you publish for the user a clear blueprint of what the model is expected to do. By *user*, I mean those who will use the API and develop larger abstractions out of it. This brings us to the topic of compositionality—APIs also need to compose. That's another issue that the algebraic approach addresses better than others. Finally, the algebra comes with a set of laws, which you can verify. This makes your model more verifiable. Let's look at these advantages in slightly more detail:

- *Loud and clear*—The algebraic approach focuses on the behaviors of the model right at the beginning. Behaviors are what the users see, and functions that implement those behaviors are what you as a modeler compose to design the domain model. You don't have the cognitive load of any object or class—just pure functions grouped together as modules that implement larger behaviors.
- *Compositionality*—A function has an algebra. Functions compose when types align—you've seen this before. When you have an API as a function, you can build larger functions just by composing from the algebra itself. You don't need to have any knowledge about the implementation of any of the composing functions. With classes and objects as in traditional OO, compositionality at the class level isn't a well-defined operation.
- *Verifiability*—By defining the laws of the algebra, you implement verifiability of your model. This makes your model robust. The set of properties that you include with the core model implementation ensures that at no stage do you violate any of them as a result of some changes in the specification.

So the main takeaway from this discussion is to appreciate the idea of algebra-based design. An algebra is a combination of a set of types, a set of functions defined with them, and a set of laws that interrelate the functions.

Note that I haven't said anything about the concrete implementation of the classes or the functions that form the model API. The initial focus is completely on the algebra and the compositionality aspects of the API. Again this is similar to what you do in algebra—if you have $y = f(x)$ and $z = g(y)$, you can always get a composed function $g(f(x))$ regardless of the concrete representation of any of f , g , x , or y . Implementation comes later in the lifecycle and takes the form of an *interpreter* of your algebra. So you have an algebra that defines the domain APIs and potentially many interpreters defining individual concrete implementations.

3.2 Defining an algebra for a domain service

Let's start with an example of an algebra-based API design for a subset of functionality from our personal banking domain. You'll keep the initial design simple enough; you'll often ignore possible complications and come back later to fix them.

Let's start with a module that defines the contract of an `AccountService`¹ with some of the operations that you plan to support. What are some example operations that you can have in such a module? Remember I mentioned earlier that unlike class-based OO, you'd have operations primarily in modules. This keeps your core domain objects lean and thin. Behaviors are no longer coupled with objects and can evolve independently using the types of the objects as the contracts of the algebra that they represent. By being decoupled from the objects themselves, this design leads to increased reusability of functions, as you'll see later. One possible operation is that of opening an account, which starts the lifecycle of the entity `Account` in the domain model. Let's define `open` with some of the arguments that you need to open an account:

```
def open(no: String, name: String, openDate: Option[Date]): ???
```

To keep things simple, I've included a few arguments to the function `open`—in reality, there will be a lot more. But what should be the return type of the function? It can be an `Account` type indicating the account that has been newly opened. But then the `open` function could fail because of validation failures, and as a good citizen of the functional programming world, you won't throw exceptions and expect users to catch them. In that case, it may make sense to return an optional data type such as `Option`. If at the same time you want to inform the user what went wrong in case of failure, you want a data type that keeps this information. You have a few options here. You can use `Either`² or `Try` from the Scala standard library. But can you identify the basic difference in returning a *value* such as `Account` from `open` as opposed to something like an `Option`, an `Either`, or a `Try` that provides an *abstraction over the evaluation*? This sounds a bit complicated, so let's take a deeper look at how functional programming offers techniques of abstraction over evaluation that lead to better compositionality.

3.2.1 Abstracting over evaluation

In this context, I'll introduce an important concept that I use throughout the book when discussing composition of domain behaviors. If you have a data type, `Account`, as the return type of `open`, you return an instantiated `Account` object if the operation succeeds. If it fails, you can either return `null` or decide to throw an exception. In all these cases, you're returning values of computation that has been evaluated, and you return the result of this evaluation. Let's compare this with returning a data type such

¹ I defined this service as a domain service in chapter 1.

² `Either` is yet another algebraic data type that models disjunction. `Either[A, B]` contains either a value of type `A` (referred to as the left projection) or a value of type `B` (referred to as the right projection).

as Option or Try. Strictly speaking, Option and Try aren't data types. They're type constructors and they model an effectful computation. Try abstracts the effect of failures, whereas Option models the effect of optionality (which means you don't have to have a value). When you return Option or Try, you're returning an abstraction over the evaluation. The caller of your function can decide whether to evaluate the abstraction by extracting the necessary value out of it or compose it with other abstractions. The advantage is that you get more compositionality, as you'll see in the examples discussed in the next section. See the sidebar "Try as an abstraction" for details on how Try not only is a means of handling exceptions, but also offers an abstraction that you can compose to build larger abstractions before evaluation for handling failures.

3.2.2 Composing abstractions

Now that you've seen that `open` will return an abstraction rather than a concrete value, you need to ensure that this abstraction is able to compose with other similar functions when you chain them together. Consider a sequence of operations that you need to perform on an account, such as opening it and performing a series of debits and credits on it. This looks like a sequence of operations that you do in imperative programming. But in the functional paradigm, you'd like to give the user the feel of imperative sequencing while doing expression-oriented programming under the hood. And this is where the returned abstraction of your function has a big role to play.

When you have a sequence of operations, what you return from each of them needs to sequence through in case of success and generate the final computation. If any of the operations fails, the entire composed operation needs to report a failure. In Scala, you typically achieve this by using for-comprehensions:

```
for {
  a <- open(..)
  b <- credit(..)
  c <- credit(..)
} yield(..)
```

Figure 3.2 shows how the composition needs to work with the computation structure that we've been talking about. The abstraction needs to support a chaining of the success path while allowing aborting the composition of operations in case of any failure.

Figure 3.2 illustrates the *monadic* model of computation,³ which you'll use to chain the methods of `AccountService` in case of sequenced operation invocation. The return type of each of the service functions needs to support this monadic model. This is the strategy that you'll adopt for composing your algebras. Using this approach, where you chain your operations, you also get an idea of what I mean when I say that a computation is a monad. Here the computation happens within a structure

³ This is just a general description of what a monadic model is—chapter 4 covers monads in detail.

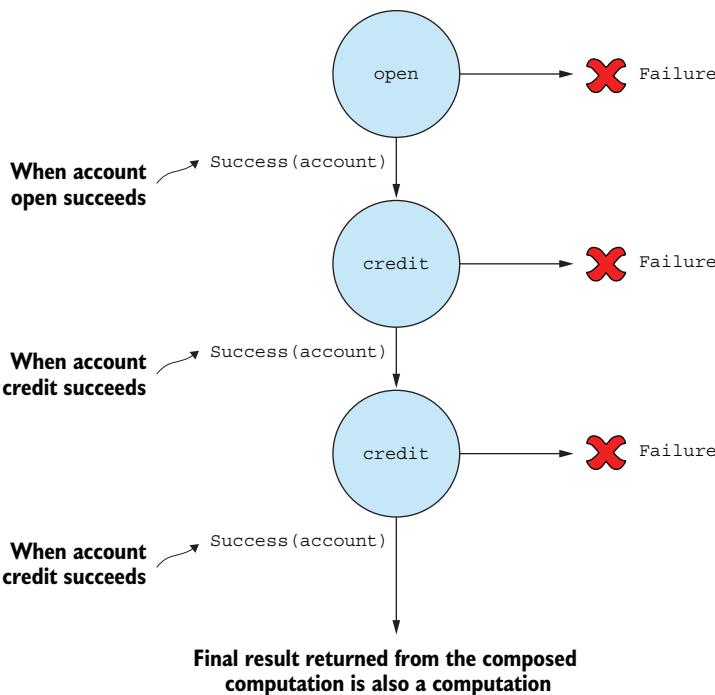


Figure 3.2 The vertical arrows denote the success path. When an operation succeeds, you have the result within the Success branch of your computation, which then flows down to the next operation. If all operations succeed, you have the final computation as the result. Note that the chaining is something that the computation offers. If at any stage the operation fails, you get a Failure branch of the computation.

that abstracts some semantics—you use a Try for AccountService APIs that abstracts the semantics of failure. I call such computations *effectful*, and monads are one way to compose effectful computations. You’ll learn more about such models in chapter 4 when we discuss other patterns of functional programming.

Try is a monad as defined in the Scala standard library.⁴ Try implements flatMap, which allows the sequencing of effects that we just discussed. Using the compositional power of Try, you can define operations composed from primitive ones, as you’ll see shortly when you implement a transfer operation from debit and credit.

By this point, you should have a good idea of how to compose algebras of domain behaviors using monads. Computationally, a monad, for all practical purposes, is an

⁴ Try violates some laws of abstraction design (as pointed out at <https://issues.scala-lang.org/browse/SI-6284>) and is therefore not a lawful monad. See more on laws of algebra in the next section. Chapter 4 discusses monads in detail.

abstraction of computation that supports a certain number of operations. Chapter 4 covers monads in detail as a generic computation structure for domain model design.

Try as an abstraction

What's the advantage of having a return type of `Try[Account]` instead of `Account` for the `open` method? When you return an account (which is an instance of the type `Account`), you return something that has already been evaluated. And these values don't compose. If you return a `Try[Account]`, you return something that abstracts the result of the evaluation. The result can be a `Failure` or a `Success`, but the abstraction gives you the power to algebraically compose the returned value with other abstractions. Take a look at the composed computation that the `transfer` function implements.

3.2.3 The final algebra of types

Now that you've seen how to define the algebra of sample domain behavior, let's look at the complete algebra of types that `AccountService` publishes. We'll end this section with an example of how to compose a coarser service method out of smaller ones by using only the algebra of the composing methods.

Some of the other operations that you can have in this module are `close` (for closing an account), `debit` (for debiting money from an account), `credit` (for crediting money into an account), and `balance` (which returns the current balance for the account). In defining these operations, you've introduced a couple of types that semantically speak the language of the domain. Right now you have no idea what these types will look like, and I haven't said anything about how you'll be implementing these types. So let's parameterize the module `AccountService` on these types. As I said earlier, that's the precise purpose of the algebra that this module and the APIs define—to decouple the contract from the implementation. The following listing shows the module along with its operations.

Listing 3.1 The AccountService module with supported operations

```
trait AccountService[Account, Amount, Balance] {
    def open(no: String, name: String, openDate: Option[Date]): Try[Account]
    def close(account: Account, closeDate: Option[Date]): Try[Account]
    def debit(account: Account, amount: Amount): Try[Account]
    def credit(account: Account, amount: Amount): Try[Account]
    def balance(account: Account): Try[Balance]
}
```

Parameterized
on types

Every function returns a `Try`. And as I mentioned earlier, there are two reasons for this. First, it helps pass on the failure information to the user. And second, it helps you sequence operations, because `Try` is a monad. Here's an example of how you can use `debit` and `credit` together and compose a new function, `transfer`:

```
def transfer(from: Account, to: Account, amount: Amount):
    Try[(Account, Account, Amount)] = for {
        a <- debit(from, amount)
        b <- credit(to, amount)
    } yield (a, b, amount)
```

Voilà! You've defined the complete implementation of a business operation just from the algebra itself. The algebra of the function `transfer` consists of the types it takes and returns and defines the business contract of the domain behavior. Note that when you talk about the algebra, you talk only about the *types* that form the contract and not the *implementation*. You have no idea what your types `Account` and `Balance` will look like or how you'll implement the behaviors `debit` or `credit`. And yet you have a complete implementation of a function that's formed from them. This is algebraic composition; you could compose the previous functions only because the types of the algebra aligned beautifully. And note how you've just specified *what* you want to do in the composed function. Some magic of the `Try` data type took care of the *how* part. In reality, what happens is that your monadic APIs that return `Try` bind the operations together. The for-comprehension syntactic sugar gives you an imperative feel of statements being executed in sequence while internally preserving all the goodness of expression-oriented programming. Now you're in a position to include your `transfer` function as part of the module definition, because it doesn't depend on the implementation of any of the functions.

What's a monad?

In functional programming, you state computations as expressions. An expression can be a primitive computation or a complex one that evolves as a result of gluing together simpler ones. When you build domain behaviors as part of your domain model, you implement combinators that evolve in a similar way. You start with simpler functions, and then using the power of higher-order functions, compose them to design larger abstractions. A monad abstracts one style of computation for building such combinator libraries.

A monad abstraction consists of the following three elements:

- A type constructor, `M[A]`, expressed in Scala as `trait M[A]` or `case class M[A]` or `class M[A]`.
- A unit method that lifts a computation into the monad. In Scala, you use the invocation of the class constructor for this purpose.
- A bind method, which sequences computations. In Scala, `flatMap` is the bind.

This implies that a monad is an algebraic structure. Any monad that has these three elements also needs to honor the following three laws:

- *Identity*—For a monad `m`, `m flatMap unit => m`
- *Unit*—For a monad `m`, `unit(v) flatMap f => f(v)`
- *Associativity*—For a monad `m`, `m flatMap g flatMap h => m flatMap {x => g(x) flatMap h}`

3.2.4 Laws of the algebra

As I mentioned earlier, one of the steps toward an algebra-based API design is to formalize the laws of the algebra. You need to explicitly record some of the invariants that your APIs have to honor. These can be generic constraints or they can be driven by rules from the domain.

Let's consider an example. One of the basic laws that you should enforce is as follows: *For all accounts, given a balance B, a successful sequence of one credit and one debit of equal amount will give back B.* Using our API definitions, you can formalize the law as a property of your model and record it as part of your test suite. You'll explore how to verify invariants of the model when we discuss property-based testing in chapter 9. Here's a sample translation of the law:

```
property("Equal credit & debit in sequence retain the same balance") =
  forAll((a: Account, m: BigDecimal) => {
    val Success((before, after)) = for {
      b <- balance(a)
      c <- credit(a, m)
      d <- debit(c, m)
    } yield (b, d.balance)

    before == after
  })
```

This code snippet uses a library named ScalaCheck (<http://scalacheck.org>), which is a popular library for property-based testing in Scala. You'll learn about this tool in chapter 9. The important point to take away from this discussion is that by using such invariants and encoding them as *verifiable* properties, you can not only document domain-model constraints but also execute them every time you build your system—yet another great aspect of algebra-based API design.

LAWS MAKE AGGREGATES CONSISTENT

As you know, the most important property of an aggregate is that it defines the consistency boundary of the abstraction. Whatever operations you do on an aggregate, it never becomes inconsistent from the model point of view. The laws of your algebra must ensure this. Many of the laws will be verified by the type system, whereas others can be checked using property-based testing discussed at the beginning of section 3.2.4.

In the example module in listing 3.1, one of the consistency guarantees is that when you close an account, the close date must be greater than the open date of the account—which means that none of the operations on an `Account` that the module `AccountService` implements can violate this invariant. The most obvious choice of a function that can potentially do so is the `close` operation. So, as part of the laws that the module has to honor, you must have one that enforces this on the account aggregate. You'll see how to enforce this when you implement the module and the `close` operation in the next section.



EXERCISE 3.1 MANY WAYS TO MODEL FAILURE

In this section, you've used Try as the computation structure to model failures. Let's explore other options and find out the relative merits of the approaches:

- Rewrite the definition of AccountService by using Option as the return type of the methods. Hint: Option also implements flatMap and can be used for chaining computations. Do you think Option is a better way to model your use case than Try?
- Use Either to model the failure of the operations. Like Try, Either is a sum type and can be used to separate the success and failure branches. Is this a better option than Try when it comes to chaining the operations in a sequence?

3.2.5 The interpreter for the algebra

The algebra defined in the previous section sums up the contract for the API for your domain model. You may now wonder, what about the implementation of the API? The idea is to decouple the implementation from the algebra itself so that you can allow multiple implementations for a single algebra. Each implementation is known as the *interpreter* of the algebra and will consist of concrete classes and functions that implement the API definitions.

Remember, when you defined the APIs, you pulled some types out of thin air (for example, Amount, Account, Balance) and parameterized AccountService on them in listing 3.1. Now it's time to provide some concrete implementation for each of those types, as shown in the following listing. And listing 3.3 provides a sample interpreter of the algebra defined in listing 3.1.

Listing 3.2 Base abstractions for your AccountService API implementation

```
type Amount = BigDecimal
def today = Calendar.getInstance.getTime
case class Balance(amount: Amount = 0)
case class Account(no: String, name: String, dateOfOpen: Date,
  dateOfClose: Option[Date] = None, balance: Balance = Balance(0))
```

Listing 3.3 The interpreter of your algebra

```
object AccountService extends AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, openingDate: Option[Date]): Try[Account] = {
    if (no.isEmpty || name.isEmpty)
      Failure(new Exception(s"Account no or name cannot be blank"))
    else if (openingDate.getOrElse(today) before today)
      Failure(new Exception(s"Cannot open account in the past"))
    else Success(Account(no, name, openingDate.getOrElse(today)))
  }
}
```

```

def close(account: Account, closeDate: Option[Date]): Try[Account] = {
    val cd = closeDate.getOrElse(today)
    if (cd before account.dateOfOpen)
        Failure(new Exception(s"Close date $cd cannot be before
➥ opening date ${account.dateOfOpen}")))
    else Success(account.copy(dateOfClose = Some(cd)))
}

def debit(a: Account, amount: Amount): Try[Account] = {
    if (a.balance.amount < amount)
        Failure(new Exception("Insufficient balance"))
    else Success(a.copy(balance = Balance(a.balance.amount - amount))))
}

def credit(a: Account, amount: Amount): Try[Account] =
    Success(a.copy(balance = Balance(a.balance.amount + amount)))

def balance(account: Account): Try[Balance] = Success(account.balance)
}

```

Figure 3.3 depicts how to separate the algebra from the interpreter. You can also have multiple interpreters for *a single algebra*. For all implementation details and completely runnable code, take a look at the downloadable code for this book.

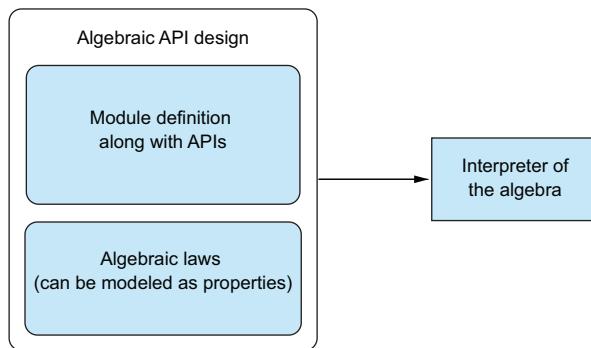


Figure 3.3 Algebra-based API design. The module contains the API definition and is usually implemented with Scala traits. The laws are implemented using properties and can be verified explicitly. The interpreter of the algebra is the API implementation and is decoupled from the definition.

3.3 Patterns in the lifecycle of a domain model

Chapter 1 presented the following three stages that make up the lifecycle of a domain object:

- *Creation*—The object is created from its components.
- *Participation*—A valid domain object interacts with other abstractions to deliver functionality in the domain.
- *Persistence*—A domain object gets written on to some persistent store.

Whenever we talk about a domain model implemented using any technology, these are mostly universal patterns. Let's consider these three lifecycle phases for the most important object in the domain of personal banking—the customer account. Whenever you walk up to the nearest bank and request the friendly teller to open an

account for you, the underlying banking system *creates* an Account object for you. It has to be a valid object containing your details as the holder of the account, and then the system assigns a valid account number as a unique identification tag to it. Once the account is created, it participates in various account-related transactions (such as debit, credit, and transfer) that directly impact your account status and balance. It can also participate in various back-office operations, such as posting interest to your account. All of these are examples of the *participation* phase, where the domain object is an intrinsic party to the banking activities. Your account details can't always reside in the volatile memory of the system—sometimes you need to *persist* accounts and transactions in a store, which also offers querying, updating, and deleting capabilities. All of these lifecycle patterns relevant for the aggregate of all domain objects constitute the overall model of your system.

This chapter covers the implementation of some of these patterns using the techniques of functional programming in general and Scala idioms in particular. You can create an object in a multitude of ways—the simplest being a direct invocation of the class constructor. But the simplest technique always has some pitfalls, and in almost all nontrivial use cases it may look naïve as well. You need to apply proper software engineering principles when designing any creational pattern for your domain model. The process of creation needs to address validation too—your objects need to be *minimally valid* on creation. You'll see what I mean by that shortly.

After the object passes through the domain validation phase, you get a fully created valid domain aggregate. As you learned in chapter 1, an *aggregate* is the complete domain entity that describes a central concept of the domain model. The example in chapter 1 was that of an Account, which undoubtedly is one of the most central actors in your model. (If you need a refresher on domain aggregates, take a break and revisit chapter 1 for more details.) Validation is an important topic—validations need to be pluggable to objects and reusable under various constraints, and they need to have specific failure semantics and a domain-friendly syntax. You'll see how function composition provides a perfect recipe for all these conditions. We'll discuss *domain validation patterns* in detail later in this chapter.

Now that you have an aggregate (which is a valid domain entity), you must be wondering what constitutes the internal implementation of an aggregate. Class-based OO techniques have taught you how to couple data elements and functions within the same modular structure and come up with a chubby-looking hierarchy of domain elements. OO aficionados call it a *rich domain model*; the degree of richness is directly proportional to the amount of structure and function that you can combine together under the same class hierarchy. In functional design, we talk about *skinny* domain objects—you pack in just the amount of structure necessary to characterize the core aspects of the abstraction. You'll use algebraic data types to model our skinny elements. The functionality will be distributed across reusable and extensible structures called *modules* that will form the algebra of the model. Each module roughly corresponds to a unit of business functionality—for example, a *Portfolio* module may

deal with various functionalities for computing and reporting client portfolios. This approach of modularization makes large models understandable by structuring in a way that feels more intuitive to the user. Also, if your language has support for first-class modules, you can compose smaller modules to build larger ones and encourage organic evolution of model functionalities. This, in turn, makes your model more reusable and maintainable.

An aggregate takes part in the various domain behaviors. You'll learn how to model such behavioral patterns in terms of abstractions that you can reuse effectively and often prove their correctness mathematically. Most of these abstractions will be *lawful*; they'll obey certain laws that prevent them from being inconsistent or erroneous at any stage of their domain lifecycle. You'll ensure they're correct and use the power of the type system as a guarantee toward correctness.

You'll also look at persistence patterns—mainly from the point of view of decoupling persistence from the main workflow of the domain. You won't explore specific database concerns or technology, but you'll ensure that your implementation is generic enough to interact with many persistent stores.

Let's start with some patterns related to the creation of domain objects and how you can present to the user a unified interface for creating families of related model elements.

3.3.1 Factories—where objects come from

Put bluntly, you create objects by using the constructors of the class if you're dealing with a class-based language. Scala is a class-based language, but you'll use its functional abstraction for creation—the algebraic data type, implemented as case classes. You've seen in earlier chapters examples of constructing objects using case classes. In this section, you'll enrich the semantics of creation with large domain models.

The more complex the model becomes, the better you need to manage your abstractions. That's one of the driving forces behind adopting specific creation strategies for your domain elements. Factories offer a unified interface for creating your objects, but the term *factory* is just part of a pattern vocabulary—it's the implementation of this pattern that varies across programming languages and paradigms. Scala provides case classes as a way to model immutable domain elements, and case classes give you a free companion object, the default factory for creation. You can use them for creating your domain elements. But there are so many other things to consider for object creation when we're talking about complex models to be deployed and managed at scale:

- How do you ensure that the factory returns a *valid* object to the client?
- Where should the validation logic go?
- What happens if the validation fails—do you throw exceptions? But as you saw in chapter 1, exceptions violate referential transparency.

You have to be smarter with construction of objects. Your factory needs to return an object that's *minimally* valid, which means it can't have invalid or inconsistent core

constituents. There can be some detailed business validations, which haven't yet been executed, but basics such as having a negative age field must not be allowed.

3.3.2 The smart constructor idiom

The standard technique for allowing easy construction of objects that need to honor a set of constraints is popularly known as the *smart constructor* idiom. You prohibit the user from invoking the basic constructor of the algebraic data type and instead provide a smarter version that ensures the user gets back a data type from which she can recover either a valid instance of the domain object or an appropriate explanation of the failure. Let's consider an example.

In our personal banking domain, many jobs may need to be scheduled for execution on specific days of the week. Here you have an abstraction—a *day of the week* that you can implement so that you can have it validated as part of the construction process. You may represent a day of the week as an integer value, but obviously it needs to honor some constraints in order to qualify as a valid day of a week—it has to be a value between 1 and 7, 1 representing a Monday and 7 representing a Sunday. Will you do the following?

```
case class DayOfWeek(day: Int) {
    if (day < 1 or day > 7)
        throw new IllegalArgumentException("Must lie between 1 and 7")
}
```

This violates our primary criterion of referential transparency—an exception isn't one of the benevolent citizens of functional programming. Let's be smarter than that. The following listing illustrates the smart constructor idiom for this abstraction. Take a look at the code and then we'll dissect it to identify the rationale.

Listing 3.4 A DayOfWeek using the smart constructor idiom

```
sealed trait DayOfWeek {
    val value: Int
}

override def toString = value match {
    case 1 => "Monday"
    case 2 => "Tuesday"
    case 3 => "Wednesday"
    case 4 => "Thursday"
    case 5 => "Friday"
    case 6 => "Saturday"
    case 7 => "Sunday"
}

object DayOfWeek {
    private def unsafeDayOfWeek(d: Int) = new DayOfWeek { val value = d }
    private val isValid: Int => Boolean = { i => i >= 1 && i <= 7 }
    def dayOfWeek(d: Int): Option[DayOfWeek] = if (isValid(d))
        Some(unsafeDayOfWeek(d)) else None
}
```

The core abstraction that models a day of the week

The companion object or the module

The published smart constructor

Let's explore some of the features that this implementation offers that make the implementation a smart one:

- The primary interface for creating a `DayOfWeek` has been named `unsafe` and marked private. It's not exposed to the user and can be used only within the implementation. There's no way the user can get back an instance of `DayOfWeek` by using this function call. This is intentional, because the instance may not be a valid one if the user passed an out-of-range integer as the argument to this function.
- The only way to get an instance of a data type representing either a valid constructed object or an absence of it is to use `dayOfWeek`, the smart constructor from the companion object `DayOfWeek`.
- Note the return type of the smart constructor, which is `Option[DayOfWeek]`. If the user passed a valid integer, then she gets back a `Some(DayOfWeek)`, or else it's a `None`, representing the absence of a value.
- To keep the example simple, `Option` is used to represent the optional presence of the constructed instance. But for data types that may have more complex validation logic, your client may want to know the reason that the object creation failed. This can be done by using more expressive data types such as `Either` or `Try`, which allow you to return the reason, as well, in case the creation fails. You'll see an illustration of this in the next example.
- Most of the domain logic for creation and validation is moved away from the core abstraction, which is the trait, to the companion object, which is the module. This is what I meant by skinny model implementation, as opposed to rich models that OO espouses.
- A typical invocation of the smart constructor could be `DayOfWeek.dayOfWeek(n).foreach(schedule)`, where `schedule` is a function that schedules a job on the `DayOfWeek` that it gets as input.

This section ends with another example from our domain that involves the creation of related objects. You'll use a more expressive type for the creation function that will return details of the failure information to the user in case the creation process fails.

What's a sealed trait?

In Scala, a trait or a class is sealed if all the subtypes of the trait or the class are in the same file as the one where the sealed trait is defined.

When you make a trait sealed, the compiler gets to know all possible subtypes that it may have and hence can reason about it. For example, if you use the trait in pattern matching and don't include all subtypes in the match, the compiler can emit a warning stating that the match is not exhaustive.

Use a sealed trait for domain elements for which the number of subtypes is fixed up front.

3.3.3 Get smarter with more expressive types

You can create various types of accounts—checking and savings—and the creation and validation logic still takes place through smart constructors within the module system. Take a look at listing 3.5; as you can see, it differs slightly from the implementation of listing 3.4. The most notable difference is that you return a more expressive type (`Try`) from the creation function. As I've said before, `Try` helps you return the reason that the construction failed, in case the client code wants to return this information upstream. With a powerful type system, you can be just as expressive with your model implementation, without compromising an iota of compile-time safety.

Listing 3.5 Creating Account using smart constructors

```

import java.util.{ Date, Calendar }
import util.{ Try, Success, Failure }

type Amount = BigDecimal
def today = Calendar.getInstance.getTime

case class Balance(amount: Amount = 0)

sealed trait Account {
    def no: String
    def name: String
    def dateOfOpen: Option[Date]
    def dateOfClose: Option[Date]
    def balance: Balance
}

final case class CheckingAccount private (no: String, name: String,
    dateOfOpen: Option[Date], dateOfClose: Option[Date],
    balance: Balance) extends Account

final case class SavingsAccount private (no: String, name: String,
    rateOfInterest: Amount, dateOfOpen: Option[Date],
    dateOfClose: Option[Date], balance: Balance)
    extends Account

object Account {
    def checkingAccount(no: String, name: String, openDate: Option[Date],
        closeDate: Option[Date], balance: Balance): Try[Account] = {
        closeDateCheck(openDate, closeDate).map { d =>
            CheckingAccount(no, name, Some(d._1), d._2, balance)
        }
    }

    def savingsAccount(no: String, name: String, rate: BigDecimal,
        openDate: Option[Date], closeDate: Option[Date],
        balance: Balance): Try[Account] = {
        closeDateCheck(openDate, closeDate).map { d =>
            if (rate <= BigDecimal(0))
                throw new Exception(s"Interest rate $rate must be > 0")
            else
                SavingsAccount(no, name, rate, Some(d._1), d._2, balance)
        }
    }
}

```

Base contract for Account

Specialization for checking and savings accounts implemented as ADTs

Smart constructors for creating accounts

```

private def closeDateCheck(openDate: Option[Date],
  closeDate: Option[Date]): Try[(Date, Option[Date])] = {
  val od = openDate.getOrElse(today)

  closeDate.map { cd =>
    if (cd before od) Failure(new Exception(
      s"Close date [$cd] cannot be earlier than open date [$od]"))
    else Success((od, Some(cd)))
  }.getOrElse {
    Success((od, closeDate))
  }
}
}

```

We discussed the smart constructor idiom earlier in a fairly detailed way with two concrete examples from our personal banking domain. You now have a clear idea of the use cases where this technique looks valuable. Here are the main takeaways that summarize this usefulness:

- The smart constructor idiom is useful when you need to instantiate domain objects that need to honor a set of constraints.
- The constructor of the class needs to be protected from users, because using it directly may lead to unsafe and inconsistent creation. The `private` specifier in the class declaration prevents direct instantiation of the case classes, whereas the `final` keyword prevents inheritance.
- The published API needs to be explicit about failure situations, and the returned data type has to be expressive enough to the user.
- For simple objects that need no explicit validation, you can directly use the class constructor. Use named arguments to make the constructor invocation expressive.

In this section, you explored the general patterns that occur frequently in the lifecycle of a domain object. You learned about creation patterns and example implementations. You'll see other patterns such as aggregates and repositories in later sections. You'll learn how implementations differ compared to similar structures that you've already encountered in OO programming. I promised you better reuse with functional programming, and you'll see how to implement aggregates and repositories using functional techniques that make them reusable, extensible, and easily testable.

3.3.4 Aggregates with algebraic data types

As we discussed earlier, an aggregate provides a single point of reference to an entity for the outside world. When you think of an `Account` as an entity, the `Account` may contain other entities such as `Address`⁵ or other value objects such as dates. But as a

⁵ As discussed in chapter 1, an `Address` can be either an entity or a value object, depending on the bounded context.

client of the API, you'd like to manipulate accounts without going into the implementation details of the other elements that make up an `Account`.

Let's consider how to implement the portfolio of a client. A portfolio consists of a collection of positions that indicate the various currency balances that a client may have as of a specific date. For example, client John Doe may have the following portfolio as of March 26, 2014:

- Account E123 has a balance of EUR 2,300.
- Account U345 has a balance of USD 12,000.
- Account A754 has a balance of AUD 3,456.

How can you model the portfolio as an aggregate so that the client of your API doesn't have to deal with individual elements such as `Position` or `Money`? Here's one way of implementing the aggregate:

```
sealed trait Account
sealed trait Currency
case class Money(amount: BigDecimal)
case class Position(account: Account, ccy: Currency, balance: Money)
case class Portfolio(pos: Seq[Position], asOf: Date)
```

Here you have an aggregate for the portfolio, with the ADT `Portfolio` as the aggregate root. There are many other elements—entities such as `Account` and value objects such as `Date`, `Money`, or `Currency`—that form the total structure of the aggregate. To the client, `Portfolio` is the central and only point of interaction, and you can ensure this by encapsulating all details of a portfolio implementation from the user. The aggregate root publishes methods that take only relevant details from the user and report appropriate information:

```
case class Portfolio(pos: Seq[Position], asOf: Date) {
    def balance(a: Account, ccy: Currency): BigDecimal = ← | Get my balance for a
        pos.find(p => p.account == a && p(ccy == ccy) ← |
            .map(_.balance.amount).getOrElse(0) | a specific currency and
    def balance(ccy: Currency): BigDecimal = ← | Get my balance for a specific
        pos.filter(_.ccy == ccy) ← |
            .map(_.balance.amount).foldLeft(BigDecimal(0))(_ + _) | currency across all accounts.
    }
}
```

You've designed a somewhat "fat" aggregate here—it contains multiple entities such as `Portfolio` (the aggregate root) and `Account`, whose portfolio is being generated. In many circumstances, this design may not scale, especially if you have a large model. It becomes difficult to enforce consistency boundaries with invariants and transactions when you have multiple entities participating in an aggregate. In that case, you can always refactor this to have `Position` contain the account number instead of the whole `Account` entity. Because an account number uniquely identifies an account, the implementation can construct an `Account` instance when required. The aggregate

thus has only one entity, `Portfolio` (which is the aggregate root), and all others are value objects.⁶

In typical OO development, you keep the related functions that manipulate a `Portfolio` within the class itself. But as you saw earlier in this chapter, with functional programming, you want to keep abstractions minimalistic and lean. The `Portfolio` ADT will contain only the minimal structure that makes up a portfolio. All associated functions go inside modules that provide API definitions, which the client can use. Refer back to section 3.1 to review how to design the algebra of modules to organize domain functions.

The major takeaway from this discussion is that an aggregate consists of (a) algebraic data types that form the structure of the entity, and (b) modules that offer the algebra of manipulating the aggregates in a compositional manner.

MODULARITY

Separating the two parts of an aggregate is an extremely important concept that you should keep in mind while designing domain models. This leads to modularity in design and a clear separation of the structure and functions of the aggregate. Don't allow the implementation structure of an aggregate to leak to the client. Even some of the functional programming techniques such as pattern matching can sometimes lead to implementation details being exposed to the client. Unless there's an important reason, ensure that all manipulations of the aggregate are done through the law-abiding algebra of modules.

Warning about pattern matching

Pattern matching with case classes is frequently used in Scala. It's syntactically convenient and expressive and readable to the user. But pattern matching has a couple of drawbacks that as a domain modeler you should be aware of:

- Patterns expose object representations and hence are considered anti-modular.
- You can't define custom patterns for matching—they're tied one-to-one with the types of case classes. Therefore, patterns aren't extensible.

In many situations, you may want to consider using Scala's Extractor pattern. For more information on why extractors are an improvement over pattern matching, take a look at the paper by Burak Emir, Martin Odersky, and John Williams.⁷

INVARIANTS

When we talk about a model and its associated elements, all of them function based on the rules of the domain. Whatever the situation, these rules can never be violated;

⁶ For more on aggregate design principles, see "Effective Aggregate Design," by Vaughn Vernon, 2011 (http://dddcommunity.org/library/vernon_2011/).

⁷ "Matching Objects with Patterns," by Burak Emir et al. (<http://lampwww.epfl.ch/~emir/written/Matching-ObjectsWithPatterns-TR.pdf>).

for instance, you can't create two accounts having the same account number, nor can you have an account with a close date earlier than the open date.

When you create aggregates, it's the responsibility of your API to ensure that such domain rules are honored. There may be some complicated business rules that you can validate after the aggregate is created. But basic validations must pass on a newly created object, and the domain must determine which of these rules qualify as *basic*. The smart constructor idiom discussed in section 3.3.2 is one of the techniques you can use to ensure that invariants are honored at the point of creation.

3.3.5 Updating aggregates functionally with lenses

Up to now we've focused on immutability and referential transparency—and now we're talking about updates. Doesn't that seem like a contradiction? Indeed, but updates are a natural phenomenon, and every domain model that you design needs to provide for the updating of domain elements.

In Scala you have vars that provide you with unconstrained mutation of objects. But for obvious reasons we aren't going to fall into that trap—it won't lead us to anywhere meaningful in the functional programming paradigm. The most common approach in functional programming is to avoid in-place mutation and instead generate a new instance of the object with the updated values. Scala case classes provide syntactic sugar for this operation. Here's an example:

```
case class Address(no: String, street: String, city: String,
  state: String, zip: String)
```

This ADT (Address) models the address of a customer in your model. If you'd like to update an attribute (say, the house no of the address), doing so is straightforward using Scala syntax:

```
Original address
val a = Address("B-12", "Monroe Street", "Denver", "CO", "80231")
val na = a.copy(no = "B-56")
a == Address("B-12", "Monroe Street", "Denver", "CO", "80231")
na == Address("B-56", "Monroe Street", "Denver", "CO", "80231")
```

copy puts in the new values in the specified fields.

New address with the changed values (it's a different object from a)

a still remains the same as the earlier one.

The diagram illustrates the functional update of an Address object. It shows the original address, its copy with a changed value, and the original address remaining the same. Annotations explain the behavior: 'copy puts in the new values in the specified fields.' points to the assignment of 'na', and 'New address with the changed values (it's a different object from a)' points to the assignment of 'na'. A central annotation 'a still remains the same as the earlier one.' points to the comparison 'a == Address("B-12", "Monroe Street", "Denver", "CO", "80231")'.

This looks cool, and it doesn't violate the axiom of immutability. But let's see if this approach scales. You'll introduce our Customer entity and give it an address:

```
case class Customer(id: Int, name: String, address: Address)
```

You'll also update the no field of the address of a specific customer by using this same copy approach:

```
val c = Customer(12, "John D Cook", a)
val nc = c.copy(address = c.address.copy(no = "B152"))
```

Do you anticipate any problems? The greater the level of nesting of the objects, the more cluttered the syntax becomes. Imagine the level of nesting of `copy` when you consider updating a deeply nested object. This situation demands better abstraction—you'd like to maintain the goodness of immutability and yet present a decent API to the user. As before, let's turn to our algebraic approach of designing abstractions.

Before giving a formal definition of the abstraction, let's try to come up with the requirements of the algebra that such an abstraction needs to satisfy. We refer to this abstraction as a *lens*.

- *Parametricity*—The lens needs to be parametric on the type of the object (let's call it `O`) that you need to update. And because each update is for a specific field of the object, the lens also needs to be parameterized on the type of the field being updated (let's call it `V`). This gives you a basic contract of a lens as case class `Lens [O, V] (...)`.
- *One lens per field*—For every object, you need to have a lens for every field. This may sound verbose, and in some cases it may feel that way. Later you'll see how to get around this verbosity by using libraries that use macros.
- *Getter*—The algebra of the lens needs to publish a getter for accessing the current value of the field. It's simply a function: `get : O => V`.
- *Setter*—The algebra of the lens needs to publish a setter. It should take an object and a new value and return a new object of the same type with the value of the field changed to the new value. It's as obvious as having a function `set : (O, V) => O`.

Summarizing these points, having an implementation of a lens ADT is as simple and obvious as the following:

```
case class Lens [O, V] (
    get: O => V,
    set: (O, V) => O
)
```

Parameterized on the object and value type

The getter, which takes an object and returns the value

The setter, which takes an object and a value and returns a new instance set to the value

The idea of a lens is nothing new. Benjamin Pierce and others have explored this data structure for solving bidirectional transformation problems such as view updates in a relational setting⁸ or transformation of tree-structured data.⁹ If you're interested in the history of lenses as a computational abstraction, take a look at these references. The

⁸ As complex it may sound, it's the problem of maintaining consistency between a view in a relational database and the underlying table of that view in case of updates. See "Relational Lenses, a Language for Updatable Views," by Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan (<http://dl.acm.org/citation.cfm?id=1142399>).

⁹ See "Combinators for Bidirectional Tree Transformations—A Linguistic Approach to the View Update Problem," by J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Allan Schmitt (<http://www.cis.upenn.edu/~bcpierce/papers/newlenses-full-toplas.pdf>).

implementation discussed in this section is a simplistic one. More sophisticated implementations of lenses include Scalaz (<https://github.com/scalaz/scalaz>) and Shapeless (<http://github.com/milessabin/shapeless>). Gérard Huet's Zipper (<http://dl.acm.org/citation.cfm?id=969872>) is another abstraction that allows functional updates to recursive data structures.

Let's take an example from our personal banking domain and see how to implement updates using a lens. Suppose you have an instance of `Address` that models the address of a bank customer and you need to change the house no attribute of the address. Given that you have a case class for `Address`, this can be easily done using the `copy` function as you did earlier. But now you have a shining lens abstraction and you'd like to use it for the update. This may sound trivial now, but remember that your goal is to have a decent API for nested updates of the house no within an address of a customer object. You want to generalize the strategy so that it scales for arbitrarily nested objects of any depth.

Let's define a lens where the object is of type `Address` and the value to be updated (the `no` attribute) is of type `String`:

```
val addressNoLens = Lens[Address, String] (
  get = _.no,
  set = (o, v) => o.copy(no = v)
)
```

How do you use this lens in practice? Here's a sample session on the Scala REPL:

```
scala> val a = Address(no = "B-12", street = "Monroe Street",
  city = "Denver", state = "CO", zip = "80231")
a: frdomain.ch3.lens.Address = Address(B-12,Monroe Street,Denver,CO,80231)
```

```
scala> addressNoLens.get(a)
```

```
scala> addressNoLens.set(a, "B-56")
res4: frdomain.ch3.lens.Address = Address(B-56,Monroe Street,Denver,CO,80231)
```

Similarly, you can define a lens for updating the `Address` field of a `Customer`:

```
val custAddressLens = Lens[Customer, Address] (
  get = _.address,
  set = (o, v) => o.copy(address = v)
)
```

You can have as many lenses as you need to do updates of individual fields. But we haven't yet addressed the nesting problem that you face with individual `copy` functions

in case classes. As is so true in functional programming, function composition once again comes to our rescue. For functions to compose, types need to align. In the previous examples, `custAddressLens` maps `Customer` to `Address`, and `addressNoLens` maps `Address` to `String`. Their types also reflect the same mapping. Do you now see the alignment? It's there waiting to be composed! But you won't implement this composition as a special case of these two specific lenses. Instead you'll define a generic `compose` function so that you don't have to repeat the same code for every pair of lenses that you compose. `compose` takes two lenses and a value, with the types of the lenses aligned for composition. The rest of the implementation is just follow-the-types to define the getter and the setter:

```
def compose[Outer, Inner, Value] ( ← Two lenses to compose.
  outer: Lens[Outer, Inner], ← Note the types.
  inner: Lens[Inner, Value]
) = Lens[Outer, Value] ( ← andThen combinator, which
  get = outer.get andThen inner.get, ← composes the two get function calls
  set = (obj, value) => outer.set(obj, inner.set(outer.get(obj), value)) ←
) ← Composed setter
```

And now you can use this `compose` function to create a bigger lens that traverses the nested data of `Customer` and jumps to update the `no` attribute within the address of the customer:

```
scala> val a = Address(no = "B-12", street = "Monroe Street",
  city = "Denver", state = "CO", zip = "80231")
a: frdomain.ch3.lens.Address = Address(B-12,Monroe Street,Denver,CO,80231)

scala> val c = Customer(12, "John D Cook", a) ← Address and Customer
c: frdomain.ch3.lens.Customer = Customer(12,John D Cook, ← defined earlier
  ↵ Address(B-12,Monroe Street,Denver,CO,80231))

scala> val custAddrNoLens = compose(custAddressLens, addressNoLens) ← Composed
custAddrNoLens: frdomain.ch3.lens.Lens[frdomain.ch3.lens.Customer, ←
  ↵ String] = Lens(<function1>, <function2>) ← lens of type
  ← [Customer, String]

scala> custAddrNoLens.get(c) ← Getter for the composed
res0: String = B-12 ← lens. Will return "B-12".

scala> custAddrNoLens.set(c, "B675") ← Setter of the composed
res1: frdomain.ch3.lens.Customer = Customer(12,John D ←
  ↵ Cook,Address(B675,Monroe Street,Denver,CO,80231)) ← lens. Will return a new
  ← Customer with no in
  ← address set to "B675".
```

One of the purposes of aggregates in the domain model is to control access to the underlying implementation through the aggregate root. The client of your API can't directly access lower-level nonroot aggregate elements. This can be done succinctly by using lens composition. Expose top-level lenses that allow transformation of lower-level objects only through the root element.

Using lenses

One common concern with lenses is determining when to use them. Isn't the `copy` function of a case class in Scala sufficient to handle functional updates? Well, like many other abstractions, a lens is scalable. For simple updates within an ADT, the `copy` function works well and is possibly the recommended approach. But consider the following scenarios:

- You need to perform updates within deeply nested ADTs. In such cases, `copy` can turn out to be unwieldy. Use a lens in such situations.
- You need to compose updates of ADTs with other abstractions. A common example is the composition with the `State` monad. A `State` monad is a way to manage application states that can change with time without using in-place mutation. A lens can be used effectively to do the update part functionally when using the `State` monad. You'll look at examples in chapter 4.

That was a brief introduction to lenses. The online code repository for the book contains a complete implementation of lenses for a domain entity `Customer`, which you can review to study how lenses fit in the overall architecture of your code base. Looking at the lens implementations of a nontrivial domain entity, you may think it's too verbose to define lenses individually for all the attributes of the entity, and wonder if all this verbosity in implementation is worth the trouble. First, you can get rid of this verbosity by using proper libraries that offer boilerplate-free implementation of lenses using Scala macros. `Monocle` (<https://github.com/julien-truffaut/Monocle>) is one such library. `Scalaz` and `Shapeless` are two excellent libraries that implement functional programming abstractions in Scala and offer lenses as part of their data structures.

LENS LAWS

Our ADT lens defines an algebra of the abstraction. As you may recall, every algebra comes with a set of laws that it needs to honor to ensure its consistency. Informally, here are the three laws that every lens needs to honor—they may seem trivial but it's worth documenting them and verifying them with property-based testing whenever you define a new one:

- *Identity*—If you get and then set back with the same value, the object remains unchanged.
- *Retention*—If you set with a value and then perform a get, you get what you had set with.
- *Double set*—If you set twice in succession and then perform a get, you get back the last set value.

In your domain model, when you define lenses to update your aggregates, don't forget to verify the laws that the lenses need to satisfy. After all, aggregates define the consistency boundary of your model. It's your responsibility as a designer to ensure

that all logical invariants are honored across all operations that you perform on an aggregate.

Once you've learned how to use property-based testing, you can verify properties for any lens that you implement. It's always a good practice to explicitly include these laws as part of your code—after all, they serve as verifiable and executable domain constraints.



EXERCISE 3.2 VERIFYING LENS LAWS

The online code repository for chapter 3 contains a definition for a `Customer` entity and the lenses for it. Take a look at `addressLens`, which updates the address of a customer, and write properties using ScalaCheck that verify the laws of a lens.

3.3.6 Repositories and the timeless art of decoupling

Chapter 1 introduced repositories. Repositories are where aggregates live, though the form or the shape may be a bit different. But you can always construct an aggregate out of your repository. How you do that depends on the structure of your repository and the impedance mismatch between the aggregate and the repository implementation paradigm. Another functionality of a repository is that you can query an aggregate from a repository. In this section, you'll learn how to do the following:

- Design and implement a repository using the features of Scala
- Organize repositories in modules using Scala traits
- Manage injecting repositories into domain services in a compositional way

Let's start with a simple API for the repository pattern. Depending on your model requirements, you can organize repositories into modules. If the model is a small one, you can implement one generic repository and have all aggregates reside within the single module. Usually, when you have to design a nontrivial domain model, you'll find yourself designing separate repositories for every aggregate and have them organized as separate Scala modules. For our current use case, you're dealing with the `Account` aggregate and want one dedicated repository module. Repositories offer certain generic functionalities that allow you to do the following:

- Fetch an aggregate based on an ID (remember, a domain entity can be uniquely identified with an ID).
- Store a complete aggregate.

You'll abstract these functions into a generic module `Repository` as follows:

```
trait Repository[A, IdType] {
  def query(id: IdType): Try[Option[A]]
  def store(a: A): Try[A]
}
```

The query may give a valid A, if found, or there can be an exception.

And you extend the generic module to one specialized to handle Account aggregates—`AccountRepository`:

```
trait AccountRepository extends Repository[Account, String] {
    def query(accountNo: String): Try[Option[Account]]
    def store(a: Account): Try[Account]
    def balance(accountNo: String): Try[Balance]
    def openedOn(date: Date): Try[Seq[Account]]
    //...
}
```

Note that you've kept the return types of the functions as `Try` to account for the failures that can happen in interacting with the repository.

Organizing repositories in modules

When you have several aggregates, organize repositories for each of them in separate modules. With this approach, you have a clear separation of algebra and subsequent implementations for units that model different entities of your problem and solution domain. And because Scala modules can be composed together, you can bundle them into a separate module when you need to use them together in a domain service. Here's an example:

```
trait PFRepos extends AccountRepository with CustomerRepository
    with BankRepository
```

After you define the algebra of a repository in a module, you can have specific implementations based on the database that you use for persisting your repository elements. The following might be the outline of an implementation based on the Redis key/value store:

```
class AccountRepositoryRedis extends AccountRepository {
    ^
    | Implementation elided
}
```

INJECTING A REPOSITORY INTO A SERVICE

Now that you have repositories organized into various modules and a way to implement multiple back ends, you need to find a way to inject these repositories into a domain service. In our domain model, the services are the main coarse level abstractions that users interact with. Coming up with a good API for the services is one of the prime criteria of good model design. By *good API*, I mean one that is succinct, concise, and most important, compositional.

A domain service needs access to the APIs of the repository—how will you model this interaction? You can start with the simplest of them all—passing the repository as an argument. Here's a module of the domain service discussed in listing 3.1 that

publishes APIs related to operating on customer accounts,¹⁰ where you injected the repository as an argument to the service methods:

```
trait AccountService[Account, Amount, Balance] {
    def open(no: String, name: String, openingDate: Option[Date],
            r: AccountRepository): Try[Account]
    def close(no: String, closeDate: Option[Date],
              r: AccountRepository): Try[Account]
    def debit(no: String, amount: Amount,
              r: AccountRepository): Try[Account]
    def credit(no: String, amount: Amount,
               r: AccountRepository): Try[Account]
    def balance(no: String, r: AccountRepository): Try[Balance]
}
```

This is the simplest approach, but at the same time the most naïve one as well. Consider the following example and try to identify the issues. This is an example of a computation on a sample customer account using the services published by `AccountService`:

```
object App extends AccountService {
    def op(no: String, aRepo: AccountRepository) = for {
        _ <- credit(no, BigDecimal(100), aRepo)
        _ <- credit(no, BigDecimal(300), aRepo)
        _ <- debit(no, BigDecimal(160), aRepo)
        b <- balance(no, aRepo)
    } yield b
}
```

The issues are as follows:

- *Verbosity*—The user of the API needs to pass the repository as a mandatory argument to the method. This is okay for a single invocation of the service method, but it's clearly verbose and boilerplate when you're composing larger abstractions out of multiple sequenced invocations (as in the previous example).
- *Coupling with the context of the API*—For every invocation of a service method, you pass the instance of the repository, which means that the repository is strongly coupled with the *context of the API*. But in reality, a repository forms the *context of the environment*—it's like a store that's there for the service method to access and do the required operations.
- *Lack of compositionality*—In this implementation, you inject the repository as a value. You can make your API far more compositional by making the injection in the context of a computation. One way to do this is to lift the argument to a curried form. That's exactly what you'll do next.

¹⁰ Note one difference in the algebra of the APIs from section 3.1. Here you don't pass an `Account`; instead you pass an account number because you'll use the repository to look up the account aggregate.

WANT COMPOSITIONALITY? CURRY IT!

The solution to all three issues is to make the repository a curried argument to the service methods. When you sequence the invocations through the for-comprehension, you can thread the repository through the computation and defer the injection until the final evaluation of the composed function. This way, you don't lose anything on the injection part and gain a lot on the compositionality of your APIs. Let's look at a sample application of this technique with our use case. First you change the algebra of the methods to lift the repository argument to a curried form (see the following listing).

Listing 3.6 Function1 as the Reader

```
trait AccountService[Account, Amount, Balance] {
    def open(no: String, name: String, openingDate: Option[Date]): AccountRepository => Try[Account]
    def close(no: String, closeDate: Option[Date]): AccountRepository => Try[Account]
    def debit(no: String, amount: Amount): AccountRepository => Try[Account]
    def credit(no: String, amount: Amount): AccountRepository => Try[Account]
    def balance(no: String): AccountRepository => Try[Balance]
}
```



All methods return Function1.

Now consider what happens when you write code like this:

```
object App {
    import AccountService._

    def op(no: String) = for {
        _ <- credit(no, BigDecimal(100))
        _ <- credit(no, BigDecimal(300))
        _ <- debit(no, BigDecimal(160))
        b <- balance(no)
    } yield b
}
```

Will this code work correctly? It will, provided you give some additional power to `Function1`, which is the type that gets threaded through the comprehension. I'm sure you now realize that the "additional power" is the power of `flatMap` defined on `Function1`. This isn't what the Scala standard library offers, but you can make `Function1` a monad and give it a `flatMap`. You'll learn the details in chapter 4 when we discuss monads. Meanwhile, if you're curious, the code for chapter 3 in the online book repository has an implementation of monads for `Function1`.

Assuming you have `flatMap` defined in `Function1`, what do you think your earlier computation `op` will return when you invoke the function?

```
scala> import App._
import App._

scala> op("a-123")
res0: AccountRepository => scala.util.Try[Balance] = <function1>
```

Note that the complete expression hasn't yet been evaluated—it's just the composed function that gets returned. Now you have to pass a repository explicitly to the returned value to evaluate the entire computation. Or you can compose it with any other computation that returns a `Function1[AccountRepository, _]` and defer the evaluation until you've built the whole computation pipeline. This latter approach is known as *building abstractions through incremental composition*, a technique used as a norm in functional programming. It's a big improvement over the approach of passing the repository as a parameter to the API.

But what if you want to compose `op` with some other computation that's not a `Function1`? It can be some other effect such as `List` or `Option`. Our current approach doesn't address this because you don't have the glue necessary to compose multiple monadic effects. That's what we discuss in the next section.

THE READER MONAD

Sometimes a computation that your function models may need some additional inputs from the environment besides the ones that you pass as explicit arguments. Instead of allowing the function to access the global namespace, the reader monad gives your function access to an environment from where it can *read* the required information. The technique just discussed does this nicely; you can use the function that an API of `AccountService` returns to pass in additional information to the computation. For all practical purposes, you can use the earlier implementation technique if you need a reader monad for your repository access.

In functional programming, it's a common practice to compose and build computation pipelines and defer evaluation until the end. The computation `op` in the earlier section is a `Function1` that may need to be composed with a monadic pipeline containing a `List` or an `Option`. You need monad transformers for this kind of composition.¹¹ But `Function1` doesn't have any transformer that can be used to stack up your computation into a monadic pipeline. Let's solve this by introducing another level of indirection. You'll wrap `Function1` into another abstraction that you'll call `Reader`. As you'll see in chapter 5, `Reader` can have a transformer, `ReaderT`, which can be used to compose other monads with the `Reader`.

```
case class Reader [R, A] (run: R => A)
```

← ADT for application of an environment,
R, to generate a value, A

Here, `Reader [R, A]` is just a wrapper for a function that runs on an environment `R` to generate an `A`. In our use case, `R` is the repository from which you'll read to get an `A`. The only thing you need to do now is implement `map` and `flatMap` in order to enable transformation and sequencing of reads in a pipeline. The following listing shows the `Reader` monad ready to be composed with our domain service APIs.

¹¹ Chapter 5 covers monad transformers.

Listing 3.7 The Reader monad

```
case class Reader[R, A](run: R => A) {
    def map[B](f: A => B): Reader[R, B] =
        Reader(r => f(run(r)))
    def flatMap[B](f: A => Reader[R, B]): Reader[R, B] =
        Reader(r => f(run(r)).run(r))
}
```

The following listing shows the service APIs, where you update the algebra to reflect this change.

Listing 3.8 The module API with Reader integration

```
trait AccountService[Account, Amount, Balance] {
    def open(no: String, name: String, openingDate: Option[Date]): Reader[AccountRepository, Try[Account]]
    def close(no: String, closeDate: Option[Date]): Reader[AccountRepository, Try[Account]]
    def debit(no: String, amount: Amount): Reader[AccountRepository, Try[Account]]
    def credit(no: String, amount: Amount): Reader[AccountRepository, Try[Account]]
    def balance(no: String): Reader[AccountRepository, Try[Balance]]
}
```

All APIs now return a Reader, to read an AccountRepository.

With the Reader in place, here's what the composition of the service APIs looks like from the user point of view. And you haven't yet passed in any concrete implementation of a repository! The client code remains the same, whether you use an explicit Reader abstraction or just `Function1` as the reader:

```
object App extends AccountService {
    def op(no: String) = for {
        _ <- credit(no, BigDecimal(100))
        _ <- credit(no, BigDecimal(300))
        _ <- debit(no, BigDecimal(160))
        b <- balance(no)
    } yield b
}
```

Nice and clean, and when you execute this snippet, you get back an instance of a `Reader`. As a final step, you invoke the `run` function of the `Reader` and pass the environment, which in our case is a *concrete* implementation of `AccountRepository`. Running `op("a123").run(AccountRepository)` will give you the balance of the account at the end of the sequence of deposits and withdrawals. Using this policy of compose up front, evaluate later also makes testing easy. Because you defer the injection of the repository until just before the evaluation stage, you can supply an alternate implementation for unit testing. If your repository is based on an enterprise back-end database, you can swap it out and plug in a simple in-memory implementation for unit

testing your services. This book’s web page features a complete implementation of AccountService along with AccountRepository injected using reader monads.

The Reader monad, in this case, yields the following benefits:

- It *decomplects* the implementation.¹²
- It makes the implementation more domain friendly by reducing boilerplate from the application code.
- It defers the injection of the concrete repository implementation at the use site of the API (instead of the definition), making the design more modular. This technique is also called *dependency injection*.¹³
- Function1 as the reader is a practical implementation of the Reader monad. Feel free to use it if you don’t need to compose with monad transformers down the pipeline.
- Because a reader decouples the computation from the environment, it makes unit testing easy. Swap out the implementation and plug in a mock one that you can unit test with easily.



EXERCISE 3.3 INJECTING MULTIPLE DEPENDENCIES

In this section’s example, you injected a repository into a domain service by using the Reader monad. In many cases, you’ll need to inject several dependencies into your domain service. For example, you may need to inject multiple repositories or another service or some configuration parameters as dependencies to a domain service.

Suggest a suitable strategy for handling injection of multiple dependencies in AccountService. One way could be to combine all dependencies into a single environment type and then pass it as a single dependency.

The main idea of this section was to show how to make repositories pluggable into service APIs from the context instead of hardcoding the concrete dependency. As you saw, using the Reader monad is one such option. We discussed two variants of implementing the Reader monad. In the options we discussed, you inject your repository as an environment to your computation. But you also can inject dependencies at a higher level of granularity (for example, in modules), and they then become available at the function levels as well. This approach leads to a slightly different architecture of your overall model, but many people use it today. Take a look at MacWire (<https://github.com/adamw/macwire>) and its philosophy of design for a simple approach to implementing this strategy.¹⁴ This book mostly uses the variants of the Reader monad pattern for injecting repositories within domain services.

¹² The term *decomplex* was first used by Rich Hickey in his presentation “Simple Made Easy” (www.infoq.com/presentations/Simple-Made-Easy/).

¹³ With a powerful language, you don’t need any framework for dependency injection. The language provides all the machinery. Here the Reader monad is our vehicle.

¹⁴ See DI in Scala: Guide by Adam Warski, 2016 (<https://di-in-scala.github.io/#manual>).

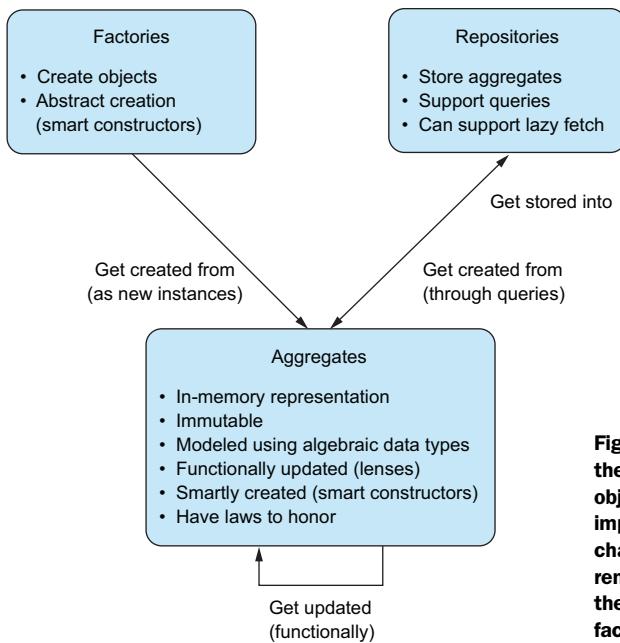


Figure 3.4 The relationship between the three lifecycle patterns of a domain object. Aggregates are the most important and active members of the chain. They'll reside in memory and render most of the functionalities of the domain model. Repositories and factories play a supporting role.

3.3.7 Using lifecycle patterns effectively—the major takeaways

We've talked a lot about aggregates and factories and how you should treat them in a functional world. Next I'll highlight the major takeaways of this powerful design pattern and provide additional pointers on alternate implementation techniques. The summary of relationships between the three patterns discussed is illustrated in figure 3.4.

- Aggregates are the primary in-memory representation of domain objects—entities as well as value objects. Design them with great care, because these data structures will be the primary point of interaction of your domain APIs.
- Abstract creation of aggregates behind factories like *smart constructors* serves two purposes—(a) the creation process details are abstracted from the user, and (b) if you have multiple subtypes of the same class, you can sometimes abstract the creation of all of them within a single API.
- Never mutate aggregates in place. Use data structures such as *lenses* that do functional updates and also offer compositional APIs. Another data structure also used to manipulate nested structures of aggregates is the *Zipper*. We won't discuss Zipper implementations here, but you can get a good idea of how they function from the original paper by its creator Gérard Huet, “The Zipper” (<http://dl.acm.org/citation.cfm?id=969872>), or by looking at implementations from Scalaz or Shapeless.
- Algebraic data types are convenient for pattern matching, as you saw in chapter 2. Pattern matching can also be used to operate on aggregates. It's better to be

conservative with this approach, though, because it can sometimes lead to inadvertent leakage of implementation. Scala provides a technique called *extractors* to address this issue.¹⁵

- Always focus on the algebraic laws that your aggregates need to honor. Keep them as explicitly verifiable properties in your test suite.

In this section, we also looked at a few generic functional programming techniques and abstractions and applied them in our domain model implementations. In this book, you'll be making more and more of these abstractions and looking at other similar structures. You must make it a point to understand how these abstractions work. If you're still not comfortable with this topic, reread this section again until you have a good grasp. Figure 3.5 summarizes the abstractions that we've discussed and maps them to the way we've used them in domain modeling.

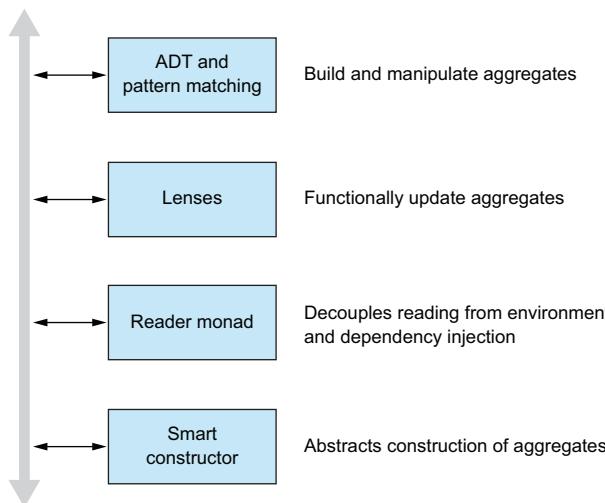


Figure 3.5 Generic functional programming abstractions and techniques mapped to the way we use them in domain modeling. You'll see more use cases for these abstractions later in this book as we discuss the other elements of domain modeling.

3.4 Summary

This is the first chapter where you learned basic patterns of domain modeling and how to approach designing APIs for your domain model by using algebraic techniques. Algebraic API design is one of the fundamental concepts that is a recurring theme in this book. Here are the major takeaways from this chapter:

- *Thinking in algebra*—You learned how to think of API design in terms of the algebra that the module defines. This is different from how you design APIs in an OO system. The initial focus is on the operations that model domain behaviors and how you collect them into modules.

¹⁵ For more details, see the paper “Matching Objects with Patterns” by Burak Emir et al. (<http://lampwww.epfl.ch/~emir/written/MatchingObjectsWithPatterns-TR.pdf>).

- *Type-driven composition*—You first identify the algebra of individual APIs and then compose them to form larger behaviors just by aligning types and using computations that bind them together.
- *Separation of concerns*—You should decouple the interpreter from the algebra; you saw a complete example. This is yet another example of separation of concerns in software engineering.
- *Aggregate as the unit of consistency*—Use the laws of the algebra for the module to enforce consistency boundaries of an aggregate.
- *Functional implementation of domain object patterns*—Implement lifecycle patterns of a domain object like factories, repositories, and aggregates by using the principles of functional programming. You saw how to design aggregates using algebraic data types, update them using lenses, design repositories, and inject a repository in a domain service using functional techniques such as reader monads. A factory is something that creates objects—we discussed the smart constructor idiom for implementing factories in Scala.

Functional patterns for domain models



This chapter covers

- Understanding functional design patterns and how they differ from OO design patterns
- Algebra as patterns
- Using monoids, the ubiquitous design pattern in functional programming
- Working with patterns for programming with effects—functors, applicatives, and monads
- Two use cases from the field that use types, algebra, and patterns to build domain models with better abstraction

The preceding chapter introduced the technique of algebraic API design and how you can think of composition of APIs before you have any implementation in your workspace. From now on when we talk about API design, we'll focus on the algebra first. In this chapter, you'll start finding patterns within the algebra of APIs. These design patterns are generic, parametric, and present themselves as artifacts that you can reuse across your domain models regardless of the nature of the domain that you're modeling.

For those of you who have used design patterns from object-oriented programming, this chapter introduces you to functional design patterns—truly reusable abstractions and not mere best practices that you’d have to reimplement every time you use them. From this chapter onward, you’ll see these computation structures reused all over our domain models. We’ll call these *design patterns*, as each is applicable to solving a specific modeling problem in a particular context. You’ll start with the algebra and implementation of monoids, one of the most widely used and most useful patterns in functional programming. Then you learn what’s called an *effectful* computation and some of the patterns for handling effects, such as applicatives and monads. Along with each of these patterns, you’ll see their implementations and the number of combinators that each offers. These combinators will help you compose these patterns generically across the elements of your domain model.

Figure 4.1 shows the flow of this chapter’s topics. This guide will help you selectively choose your topics as you sail through the chapter.

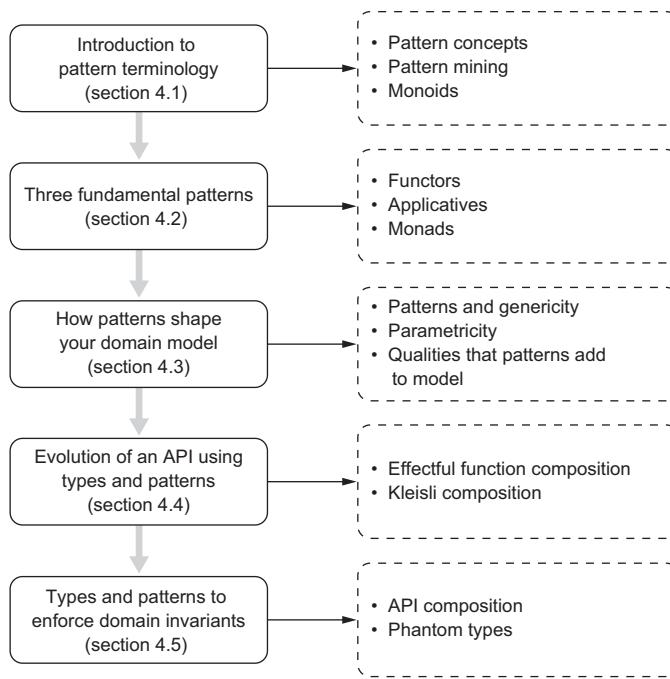


Figure 4.1 The progression of this chapter’s sections

By the end of this chapter, you should be able to discover areas of your model where one or more of these patterns make sense.

4.1 Patterns—the confluence of algebra, functions, and types

The most commonly used definition of a software pattern today is from James Coplien in *Software Patterns* (SIGS, 1996), and inspired by the works of the great architect Christopher Alexander:

A pattern is a piece of literature that describes a design problem and a general solution for the problem in a particular context.

Coplien talks about three things: a problem, a solution, and a context. Alexander, in *The Timeless Way of Building* (Oxford University Press, 1979), is more succinct when expressing a pattern as a three-part rule:

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

This definition indicates that a pattern has repeatability: The solution tends to repeat within the context. Judging from the experience of object-oriented programming, where the term *design pattern* has mostly been institutionalized, this *repeatability* comes with little *reusability*. Object-oriented design patterns are in reality design concepts (or rather a standard vocabulary) that you have to implement separately every time you have a problem and a context. Every time you need to implement a Bridge design pattern¹ to decouple an abstraction from its implementation, for example, there's nothing that you can reuse across implementations. Every bridge comes with its own set of interfaces and implementations that you need to wire together in specified inheritance hierarchies.

The functional patterns covered in this chapter offer much more reusability than OO design patterns. Each functional pattern has two distinct parts:

- Completely generic and reusable code—we call it the *algebra*²—which is invariant across all contexts where you use the pattern.
- A context-specific implementation that varies across all instances where you apply the pattern. We call it the *interpreter* of the algebra.

As you'll see in this chapter, it's the first part that shines in building reusable domain model components.

¹ See *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et. al (Addison-Wesley Professional, 1994).

² If you need a refresher on what is meant by *algebra*, I discuss it in chapter 3, section 3.1, in the context of algebraic API design.

If a pattern is reusable across contexts, it has to be generic enough to abstract common behavior that repeats across contexts. As an example, consider the following abstraction:

```
trait Monoid[T] {
    def zero: T
    def op(t1: T, t2: T): T
}
```

A monoid is parameterized on a type T . This already makes it generic with respect to the type for which you can define a monoid. And it offers the following contract as part of its algebra:

- An operation that we name `zero`, which serves as the identity element to the next function `op`. This means when you invoke `op` with `zero` as one of the arguments, it will always return the other argument.
- An operation that we name `op`, which is binary and associative.
- Both the identity part of `zero` and the associativity part of `op` are enforced by the monoid laws, which state the following:
 - *Left identity*— $\text{op}(\text{zero}, t) == t$
 - *Right identity*— $\text{op}(t, \text{zero}) == t$
 - *Associativity*— $\text{op}(\text{op}(t1, t2), t3) == \text{op}(\text{op}(t1, t2), t3)$

The definition of a monoid along with the laws that it satisfies forms the algebra. And it's this algebra that's reusable across *all* contexts. If you're dealing with a specific context of banking models, you can define an implementation of the preceding algebra that publishes `Money` to be a monoid. The algebra is immutable; you just have to interpret `zero` and `op` based on the context of `Money`.

In the definition of a monoid, note there's nothing specialized for any type. The algebra is universally quantified over a type T . You know this concept, which was discussed in earlier chapters. It's the quality of *parametricity*. Good patterns in functional programming are parametric on types, and the reusability is driven by the algebra that they define. You'll explore more of this in section 4.3, which covers how patterns can shape your domain model.

4.1.1 Mining patterns in a domain model

The patterns covered in this section rely on the virtues of parametricity. The goal is for you to explore domain behaviors and identify the commonalities that can be extracted as reusable patterns. The approach that you'll take to discover such patterns within your code is the following:

- Pick up specific use cases that you can implement using application-specific code
- Discover the commonality in structure and behavior that these specific computations have

- Identify existing generic abstractions (the patterns, specifically the algebra³) that can be specialized to model the preceding application-specific computation
- Replace application-specific code with instances of generic patterns

Abstractions in a domain-model implementation often have some commonality of behaviors along with their differences. In our domain of personal banking, consider a computation that filters high-value transactions on a daily basis or one that computes the aggregate balance of customer accounts. Although the details of computation are different in the two cases, some similarity still exists, as you fold over the collections of transactions or accounts or do a binary operation with pairwise elements in both cases. A functional pattern helps you unify the commonalities through the algebra, while allowing you to abstract over the differences with context-specific implementations. Instead of talking more in terms of abstract explanations, let's look at a concrete example from our domain and try unearthing some patterns.

4.1.2 Using functional patterns to make domain models parametric

Here's a sample use case from our domain of personal banking that implements back-office functionality.⁴ Clients perform transactions in the form of debits and credits, all of which are logged in the system for auditing and other analytical requirements. You've seen how to manage a client balance as an attribute of an account. Here you'll consider only transactions and balances and try to implement functionality that allows back-office users to compute various aggregates on transactions executed on a client account. More specifically, you'll implement the following behaviors in this part of our model:

- Given a list of transactions, you'll identify the highest-value debit transaction that occurred during the day. Typically, these values may be highlighted as exceptions for auditing purposes.
- Given a list of client balances, you'll compute the sum of all credit balances.⁵

All implementations are simple from a domain logic point of view, because the purpose of the implementation is to identify programming patterns in FP and not come up with robust, industry-standard models.

IDENTIFYING THE COMMONALITY

So far, in all earlier examples you considered a simple representation of an amount as `BigDecimal`. But in real-life banking, you always need to associate a currency with any amount you specify. So it's time to enrich this part of the model; here we go with our new `Money` model that has both the amount and the currency tagged with it. Not only that, but let's say you ask how much money I have. I check my wallet and say I have 120 U.S. dollars and 25 euros. This means our money abstraction should be able to handle

³ Remember, as I said in the preceding section, the algebra is the reusable part of the pattern.

⁴ The implementation of this example is inspired by Runar's response to this question on StackOverflow: <http://stackoverflow.com/a/4765918>.

⁵ A credit balance means a positive balance. You'll ignore a negative balance (also called a debit balance).

denominations in multiple currencies as well. The following listing contains Money and the other base abstractions that you'll use to define the algebra of your module.

Listing 4.1 Base abstractions for defining Transaction and Balance

```

sealed trait TransactionType
case object DR extends TransactionType
case object CR extends TransactionType

sealed trait Currency
case object USD extends Currency
case object JPY extends Currency
case object AUD extends Currency
case object INR extends Currency

case class Money(m: Map[Currency, BigDecimal]) {
    def toBaseCurrency: BigDecimal = //...
}

case class Transaction(txid: String, accountNo: String, date: Date,
    amount: Money, txnType: TransactionType, status: Boolean)

case class Balance(b: Money)

```

Transaction type—can be debit (DR) or credit (CR)

Currency and its enumerations

Money abstraction. You have a Map to encode denominations of multiple currencies.

Transaction that clients make in a bank

Balance of a client

Let's say the behaviors that you'll define belong to a particular module (for example, Analytics). Listing 4.2 presents the algebra of the module along with a sample interpreter.

NOTE Some details of the implementation aren't present in the following listing, but that shouldn't prevent you from understanding its essence. The full runnable source can be found in the online code repository of the book.

Listing 4.2 Algebra and implementation of module Analytics

```

trait Analytics[Transaction, Balance, Money] {
    def maxDebitOnDay(txns: List[Transaction]): Money
    def sumBalances(bs: List[Balance]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {
    def maxDebitOnDay(txns: List[Transaction]): Money = {
        txns.filter(_.txnType == DR).foldLeft(zeroMoney) { (a, txn) =>
            if (gt(txn.amount, a)) valueOf(txn) else a
        }
    }

    def sumBalances(balances: List[Balance]): Money = {
        balances.foldLeft(zeroMoney) { (a, b) =>
            a + creditBalance(b)
        }
    }

    private def valueOf(txn: Transaction): Money = //...
    private def creditBalance(b: Balance): Money = //...
}

```

Algebra of the module

Ordering defined on Money is elided. Check the online code repo.

Addition of Money elided. Check online repo for details.

valueOf finds the monetary value of a transaction.

Returns the balance only if it's a credit balance; otherwise, returns 0.

In the implementation of the `maxDebitOnDay` and `sumBalances` behaviors, do you see any similarities that you can refactor into more generic patterns? Let's list some here:

- Both implementations *fold* over the collection to compute the core domain logic.
- The folds take a unit object of `Money` as the seed of the accumulator and perform a binary operation on `Money` as part of the accumulation loop. In `maxDebitOnDay`, the operation is a comparison; in `sumBalances`, it's an addition. They are different, but both are associative and binary.

I'm sure you see where we're heading—the monoid land. This is the most important part of this exercise: to look at the pattern and identify the algebra that it fits into. It won't be a direct fit every time. Sometimes you may have to tweak your implementation to make it fit. But it's all worth it. Instead of implementing a bug-ridden variant of the existing algebra, you should always reuse it. These patterns have been refined through the years by experts and field-tested in various production implementations. The next step is to unify these two seemingly different operations by using the algebra of a monoid.

ABSTRACTING OVER THE OPERATIONS

The next step is to define an instance of `Monoid` for `Money`. Because you've defined `Money` in terms of a `Map`, you need to first define `Monoid[Map[K, V]]` and then use that to define `Monoid[Money]`. In fact, you need to define two instances of `Monoid[Money]` because you have two different requirements of operation in `maxDebitOnDay` and `sumBalances`; the former needs an instance based on comparison of `Money` and the latter needs one for addition of `Money`. Here, for brevity, I'll show the latter one; the one based on comparison is a bit verbose and is implemented in the code base in the online code repository.

```
final val zeroMoney: Money = Money(Monoid[Map[Currency, BigDecimal]]).zero)

implicit def MoneyAdditionMonoid = new Monoid[Money] {
    val m = implicitly[Monoid[Map[Currency, BigDecimal]]]
    def zero = zeroMoney
    def op(m1: Money, m2: Money) = Money(m.op(m1.m, m2.m))
}
```

Map[K, V] is a monoid if V is a monoid. Here BigDecimal is a monoid.

Listing 4.3 shows the implementation of the `Analytics` module that uses a monoid on `Money`.⁶ This is the first step toward making your model more generic. The operation within the fold is now an operation on a monoid instead of hardcoded operations on domain-specific types.

⁶ This listing (as well as many others in this chapter) contains advanced Scala idioms and syntaxes. If you need to brush up, refer to *Programming Scala* by Dean Wampler and Alex Payne (O'Reilly Media, 2014). For functional programming idioms, refer to *Functional Programming in Scala*.

Listing 4.3 Abstracting operations through a monoid

```

trait Analytics[Transaction, Balance, Money] {      ←
  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money
  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money = {
    txns.filter(_.txnType == DR).foldLeft(m.zero) { (a, txn) =>
      m.op(a, valueOf(txn))
    }
  }

  def sumBalances(balances: List[Balance])(implicit m: Monoid[Money]): Money =
    balances.foldLeft(m.zero) { (a, bal) => m.op(a, creditBalance(bal)) }

  private def valueOf(txn: Transaction): Money = ...
  private def creditBalance(b: Balance): Money = ...
}

```

Both functions now need an implicit Monoid for Money. This will be used in the implementation to transform specific operations on Money to that of a Monoid. This makes those operations reusable for any Monoid, not only Money.

Scala tip!

In Scala, you need to pass an implicit object explicitly with every function that needs it. Note how we've done the same for the functions implemented in listing 4.3. This is called the *explicit dictionary-passing technique*: You pass the dictionary of implicits along with the function definition. The advantage of this approach is that you can define multiple instances of `Monoid[Money]` (in fact, you'll do that here), and the compiler will pass the appropriate one to the function. But you need to ensure through the order and scope of declaration of implicits that the correct one gets picked by the compiler. If you have multiple implicits in scope, you have to pass the one you want explicitly.⁶

In our example, you'll need two instances, one for `maxDebitOnDay` that compares two instances of `Money`, and one for `sumBalances` that does an addition of `Money`. And you'll have to pass the appropriate instance when you invoke the operation.

ABSTRACTING OVER THE CONTEXT

In the preceding implementation, the first thing that stands out is that in both `maxDebitOnDay` and `sumBalances`, the action within the `fold` is curiously similar. In both cases, you've abstracted over the operation of the monoid that you passed.

⁷ For more details of explicit dictionary-passing techniques in Scala, see the paper “Type Classes as Objects and Implicits” by Bruno Oliveira et. al at <http://ropas.snu.ac.kr/~bruno/papers/TypeClasses.pdf>.

Because of this abstraction, the code is more generic and needs lesser knowledge of the specific domain elements.

If you squint hard at both functions, you can see another similarity. In both cases, you fold over a collection after mapping through a function that generates a monoid. For `maxDebitOnDay`, you map using `valueOf`, which is `Transaction => Money`, and for `sumBalances` you use `creditBalance`, which is `Balance => Money`. And `Money` is a monoid.⁸ If the collection has elements that themselves are monoids, you need not do any mapping (or rather you can map with an identity function). In summary, what you're doing in both functions is, given a collection `F[A]`, which can be folded over, you do a fold on `F[A]`, where either `A` is a monoid or can be mapped into one. The only property of the collection that you need is its ability to be folded over. So you can make your collection still more generic (and less powerful) by defining it to be a `Foldable[A]`; you don't need the richness of a `List[A]` to implement what you need here. Here's the algebra of your `Foldable` type constructor:

```
trait Foldable[F[_]] {
  def foldl[A, B](as: F[A], z: B, f: (B, A) => B): B
  def foldMap[A, B](as: F[A])(f: A => B)(implicit m: Monoid[B]): B =
    foldl(as, m.zero, (b: B, a: A) => m.op(b, f(a)))
}
```

The function `foldMap` does exactly what I said before: folds over a collection `F[A]`, where `f: A => B` generates a monoid `B` out of `A`. And `f` can be an identity if `A` is a monoid. So, given a `Foldable[A]`, a type `B` that's a monoid, and a mapping function between `A` and `B`, you can package `foldMap` nicely into a combinator that abstracts your requirements of `maxDebitOnDay` and `sumBalances` (and many other similar domain behaviors) without sacrificing the holy grail of parametricity. *And this is the second step toward making your model more generic using design patterns: You've abstracted over the context, the type constructor of your abstraction.*

```
def mapReduce[F[_], A, B](as: F[A])(f: A => B)
  (implicit fd: Foldable[F], m: Monoid[B]) = fd.foldMap(as)(f)
```

And now each of your module functions becomes as trivial as a one-liner:

```
object Analytics extends Analytics[Transaction, Balance, Money] {
  def maxDebitOnDay(txns: List[Transaction])
    (implicit m: Monoid[Money]): Money =
    mapReduce(txns.filter(_.txnType == DR))(valueOf)

  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money =
    mapReduce(bs)(creditBalance)
}
```

The complete runnable code of this entire exercise can be found in the online code repository for the book.

⁸ When I say `A` is a monoid, I mean that `A` is a type that has a monoid instance defined.

In summary, what did you gain from using the design patterns such as `Monoid` or `Foldable`?

- *More generic*—The domain behaviors are now implemented in terms of a completely generic `mapReduce` function, which raises the abstraction level of your model. And because `mapReduce` is completely generic, it can be reused as a solution to other similar problems. Through the exercise of pattern mining, you've discovered a generic abstraction (`mapReduce`) formed by unifying the algebras of two of our functional patterns, the `Monoid` and the `Foldable`.
- *More abstract*—You've been able to abstract over the operation by using the `Monoid` design pattern, and over the type constructor by using the `Foldable` pattern. This makes the model more modular, and easier to manage and verify.
- *Parallelizable*—Finally, because a monoidal operation is associative, it can be parallelized (all associative operations are), computed incrementally, and cached.⁹

4.2 Basic patterns of computation in typed functional programming

When you have a typed functional programming language, the most important aspect of domain modeling is the organization of your computation structures using the type system so that you can achieve maximum compositionality within your model. Remember, abstractions compose when types align; otherwise, you're left with islands of artifacts that you can't reuse and compose. There has been quite a bit of research on this, especially in the Haskell community, and people have found that building abstractions through composition of typed functions can be mapped closely to composing morphisms in the mathematical world of category theory. Many of the patterns in category theory, such as functors and monads, have found their close counterparts in the world of typed functional programming. This book doesn't cover category theory but touches on this subject to emphasize that the patterns in this section have their roots in the mathematical basis of categories and morphisms.

The patterns covered in this section form the basis of effectful computing in functional programming. They offer abstractions for handling *effects* within your domain model.¹⁰ When you design a domain model in real life, you'll see that few operations are *intrinsically* pure. In most behaviors, you need to implement effects such as handling optional values, exception tracking, and IO. Using the patterns in this section will help you design a pure and referentially transparent user interface on top of the effectful computations. You saw similar examples in chapter 3 of how to handle exceptions in Scala by using the abstraction of `Try` in designing a functional interface of the `AccountService` module. But `Try` was a special case; in this chapter, you'll learn how to generalize effects algebraically. You'll see generic patterns such as functors, monads,

⁹ For more on parallelizability, have a look at "Parallel Programming with List Homomorphisms," by Murray Cole in *Parallel Processing Letters* 5:191-203, 1995.

¹⁰ You saw how to deal with effectful computations in chapter 2.

and applicatives that develop a generic computation structure for abstracting and composing such effects. At the end of this section, you'll see how to express effects generically in your domain model and how to compose them to form larger effects.

4.2.1 **Functors—the pattern to build on**

Have you ever looked at the signatures of `map` in some of the classes of the Scala standard library such as `List`, `Option`, or `Try`? If you haven't, the following snippet shows them all in one place:

```
// map for List[A]
def map[B](f: (A) => B): List[B]

// map for Option[A]
def map[B](f: (A) => B): Option[B]

// map for Try[A]
def map[B](f: (A) => B): Try[B]
```

A pattern repeats across all of them. The functions have the same signature if you ignore the context of the application. In terms of functional programming, you call this context an *effect*. `List[A]` provides an effect of repetition of elements of type `A`. `Option[A]` models the effect of uncertainty, where the element of type `A` can either exist or not. `Try[A]` provides the effect of reification of exceptions. So if you can abstract the effects out of the signatures, all you have is a function, `map`, that lifts the function `f` into an effect `F[_]`. Each of these effects is modeled using a type constructor—hence `F[_]` and not a simple `F`. You can extract this computation structure into a separate trait of its own and call it the `Functor`:

```
trait Functor[F[_]] {
  def map[A, B](a: F[A])(f: A => B): F[B]
}
```

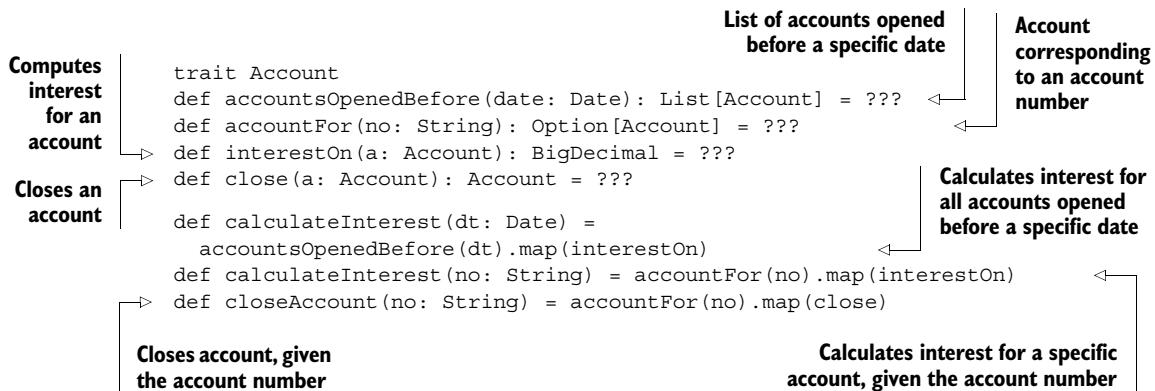
Because `List`, `Option`, and so forth all share this computational behavior, you can make them functors by implementing an interpreter of the functor algebra for each of them:

```
def ListFunctor: Functor[List] = new Functor[List] {
  def map[A, B](a: List[A])(f: A => B): List[B] = a map f
}

def OptionFunctor: Functor[Option] = new Functor[Option] {
  def map[A, B](a: Option[A])(f: A => B): Option[B] = a map f
}
```

You can say that a functor abstracts the capability of mapping over a data structure with an ordinary function. Now you may ask, what advantage does that bring in the context of domain modeling? The answer is simple: It generalizes your model and makes some

parts of your model behavior reusable across components. Let's consider an example; say you have the following functions in a module of your implementation:



In the functions `calculateInterest` and `closeAccount`, you map over the concrete, application-specific data structures by using the function supplied to compute the interest or close the account, respectively. Using functors, you can extract this mapping behavior into a common generic function parameterized over a type that provides a functor implementation. Being parametric, this function makes your code less specific to domain types and gives you a reusable function that can be used anywhere you want to map over a functor. This is exactly the scenario illustrated previously in figure 4.1. Just replace the context of Monoid with a Functor and you have this exact use case depicted there:

```
def fmap[F[_], A, B](fa: F[A])(f: A => B)(implicit ft: Functor[F]) =
  ft.map(fa)(f)
```

You can now use `fmap` to implement your domain-specific behaviors:

```
def calculateInterest(dt: Date) =
  fmap(accountsOpenedBefore(dt))(interestOn)
def calculateInterest(no: String) =
  fmap(accountFor(no))(interestOn)
def closeAccount(no: String) = fmap(accountFor(no))(close)
```

Functors give you the power to map over pure functions, and that's all they offer. Not surprisingly, soon you'll find that only a small fraction of your use cases map over pure functions. The main utility of functors is that they provide a gateway to the next patterns of effectful computation—the applicative functors and monads.

4.2.2 **The Applicative Functor pattern**

A functor gives you the capability to lift a pure function of one argument into a specific context. Look at the definition of `map` in the trait `Functor`. The pure function `f`:

$A \Rightarrow B$ is lifted into the context of F , transforming $F[A]$ to $F[B]$. The context here is the *type constructor* F , also known as the *effect* of the computation. But what happens when you have a function of two arguments that you'd like to lift into a context? Or you can even generalize and think of functions of any arity to be lifted into some context. The Applicative Functor design pattern is a solution to this generic problem.

To illustrate, let's take a sample use case from our domain and see how to apply the pattern to come up with a good functional implementation. The use case is that of processing validations for some of our domain elements.

Consider the Account entity that you modeled in chapter 3; you used smart constructors to implement the factory that created accounts of a specific type.¹¹ Here's one of the smart constructors that gave you an instance of a SavingsAccount from the passed-in arguments:

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): Try[Account] = {

  closeDateCheck(openDate, closeDate).map { d =>
    if (rate <= BigDecimal(0))
      throw new Exception(s"Interest rate $rate must be > 0")
    else
      SavingsAccount(no, name, rate, Some(d._1), d._2, balance)
  }
}
```

Besides the construction part, the smart constructor performs a validation on the consistency between the account open and close dates. There could be other validations too—for example, validating whether the account number is of a specific format or whether the supplied interest rate is a valid one. In general, validation logic can be complex, and you may have to invoke multiple validations before you return a valid aggregate to the user. Using the imperative style of programming, you can use nested conditional constructs toward this end. But in the functional paradigm, we prefer expression-oriented programming.¹² You should have all validations composed together as an expression, which returns a valid constructed aggregate when all of them are satisfied.

Instead of having a formal definition and implementation up front, let's try to form an understanding of how our composite validation use case evolves into such an abstraction and how to generalize it as a design pattern. Such an exploratory approach not only gives you an idea of what the model should look like, but also how to build one incrementally from the bits and pieces of your requirements. As usual,

¹¹ Chapter 3 discussed the smart constructor idiom.

¹² Chapters 1 and 2 cover expression-oriented programming.

you'll use an algebraic model to come up with the appropriate API. The requirements are twofold:

- Every validation function needs to return a context that will contain either the validated object or an error indicating why the validation failed.
- These contexts will then be processed either to construct the desired instance of a `SavingsAccount` or report an error to the client.

By introducing the context that abstracts the result of the validation, you've already achieved one level of separation between the construction of `SavingsAccount` and ensuring its conformance with the rules of the domain. Let's call the context `Validation[E, A]`; it will be a type constructor because it has to contain either the validated object, `A`, or an error message, `E`. You can assume that the error message will be a `String` and simplify your context to `Validation[String, A]`. So here you have the algebra of validations along with a type alias for convenience:

```
type V[A] = Validation[String, A]

def validateAccountNo(no: String): V[String]
def validateOpenCloseDate(openDate: Option[Date], closeDate: Option[Date]): V[(Date, Option[Date])]
def validateRateOfInterest(rate: BigDecimal): V[BigDecimal]
```

You know nothing about the implementation of `Validation` so far—just a placeholder for the type and how it fits into the algebra of the validation functions. After you invoke all three of the validation functions, you get three instances of `V[_]`. If all of them indicate a successful validation, you extract the validated arguments and pass them to a function, `f`, that constructs the final validated object. In case of failure, you report errors to the client. This gives the following contract for the workflow—let's call it `apply3`.¹³

```
def apply3[V[_], A, B, C, D](va: V[A], vb: V[B], vc: V[C])
  (f: (A, B, C) => D)
  : V[D]
```

The diagram illustrates the `apply3` function contract. Three arrows point from `V[A]`, `V[B]`, and `V[C]` to a central box labeled "The processing function". Inside this box is the expression `(f: (A, B, C) => D)`. Another arrow points from this box to the result `V[D]`. Labels with arrows identify the components: "The input contexts" points to the three `V` types, "The processing function" points to the `f` expression, and "Validated output object in the same context" points to the final `V[D]` type.

There's another way of looking at this function, which will give you a different understanding of the abstraction:

```
def lift3[V[_], A, B, C, D](f: (A, B, C) => D)
  : (V[A], V[B], V[C]) => V[D]
  = apply3(_, _, _)(f)
```

Here you lift the pure function, `f`, into the context, `V`. Think about a situation where you'll prefer this variant instead of using `apply3`. Exercise 4.1 explores this in more detail.

¹³ `apply` indicates that you're applying the contexts to the function. And 3 indicates the number of contexts.



EXERCISE 4.1 ALGEBRAIC THINKING

In this exercise, you'll explore aspects of algebraic design and see how one form of the algebra can sometimes be more useful than an alternative representation of the same algebra. Both the definitions of `apply3` and `lift3` in the current section achieve the same goal. They take a *pure* function of three arguments and help you use the function along with three validation contexts, one per argument. But the usages are different and will give you a different perspective of how to apply the function to the contexts.

Take a hard look at the signatures of `apply3` and `lift3`. `apply3` takes as inputs all three validation contexts and the pure function `f` of three arguments. It applies the function (just as the name suggests) to the three contexts, `V[A]`, `V[B]`, `V[C]`, and gives you back the resultant context `V[D]`. The result is the final context *evaluated*.

`lift3`, on the other hand, doesn't include the *application* of the function `f` as part of its implementation. Rather it returns an *abstraction*, which when evaluated will give you the resultant validation context. It's curried on the context arguments.

Explain the pros and cons of the two approaches and which one you'd prefer under what circumstances.

Hint: Read section 3.2.1 in chapter 3 and consider the advantages that an abstraction brings over an early evaluation. `lift3` composes better than `apply3`.

You don't yet have the implementation of `apply3`. But assuming you have one, let's see how the validation code evolves out of this algebra and plugs into the smart constructor for creating a `SavingsAccount`:

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): V[Account] = {
  apply3(
    validateAccountNo(no),
    validateOpenCloseDate(openDate, closeDate),
    validateRate(rate)
  ) { (n, d, r) =>
    SavingsAccount(n, name, r, d._1, d._2, balance)
  }
}
```

The three contexts of validation

The function that extracts information from the contexts and constructs a valid `SavingsAccount`

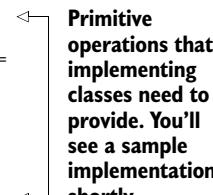
validateOpenCloseDate returns a `Tuple2`, and you access the two members using `d._1` and `d._2`.

`apply3` nicely fits this use case. But you need to generalize the entire workflow into an abstraction that can have a broader application. After all, `apply3` or `lift3` aren't APIs specific to validating a bunch of fields. Where will `apply3` or `lift3` live? Right now you've kept them in a global namespace and parameterized them on the context type constructor `V[_]`. Let's put them into a module and unearth the pattern inside.

You've just discovered the *Applicative Functor* pattern of functional programming. It's a useful pattern when you deal with effects in functional programming called *applicative effects* (soon you'll see another type of effect, the monadic effect). As the name suggests, an applicative functor builds additional capabilities on top of what a functor offers, and the term *applicative* refers to the way the effects are applied. If you look at `apply3`, you'll realize that the effects are sequenced through *all* the arguments, regardless of the result that each may produce. For example, in the preceding application, all three validation functions will be executed regardless of the success or failure of any of them. *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2014) has all the details of this pattern, which I won't repeat here. For details on the implementation of applicative functors, have a look at Scalaz (<https://github.com/scalaz/scalaz>). For readers not yet initiated to Scalaz, the next sidebar highlights some features of the library.

Listing 4.4 The Applicative Functor trait (simplified)

```
trait Applicative[F[_]] extends Functor[F] {
    def ap[A, B](fa: F[A])(f: A => B): F[B]
    def apply2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
        ap(fb)(map(fa)(f.curried))
    def lift2[A, B, C](f: (A, B) => C): (F[A], F[B]) => F[C] =
        apply2(_, _)(f)
    def unit[A](a: => A): F[A]
}
```



Primitive operations that implementing classes need to provide. You'll see a sample implementation shortly.

After you have the `Applicative` trait in place, provide an instance of `Applicative` for `Validation`.¹⁴ And you have the implementation of `savingsAccount` with the validation logic implemented with applicative effects.

```
val av: Applicative[Validation] = ...
def savingsAccount(no: String, name: String, rate: BigDecimal,
    openDate: Option[Date], closeDate: Option[Date],
    balance: Balance): Validation[Account] = {
    // ...
    av.apply3(
        validateAccountNo(no),
        validateOpenCloseDate(openDate, closeDate),
        validateRate(rate)
    ) { (n, d, r) =>
        SavingsAccount(n, name, r, d._1, d._2, balance)
    }
}
```



You invoke apply3 from the Applicative instance of V.

¹⁴ `Applicative` for `Validation` needs a partial application of types. In fact, it will have a type of `Applicative[({type a[x] = Validation[E,x]})#a]`. It's easier to write it as an `Applicative` for `V`, which is `Validation[String, A]`. *Functional Programming in Scala* provides more details on partial type application in Scala.

A complete production-ready Applicative implementation comes with the library Scalaz. The code repository that accompanies this book has a full implementation of the Account validation in the smart constructors using the Validation abstraction that Scalaz offers. It offers a generalized builder for Applicative that helps you implement lifting functions of arbitrary arity into the context of an applicative functor.

Scalaz—a brief description

Scalaz is a library that implements pure functional abstractions in the Scala programming language. It's similar in spirit to the abstractions that Haskell offers and uses the power of parametric polymorphism for composing and extending them. The following is a concise list of features that Scalaz offers:

- The base abstractions of Scalaz are type classes, which allow ad hoc polymorphism for extension. Scalaz offers type classes for almost every abstraction that you may need to do pure functional programming. For a general idea of the type class concept and the hierarchies, take a look at Typeclassopedia.
- In addition to a bunch of type classes, Scalaz offers implementation of purely functional data structures such as Finger Search Tree, Difference List, NonEmptyList, Lenses, and Zippers.
- A way to control effects as in Haskell.
- Monad transformers such as ReaderT, WriterT, and StateT, which you can use to compose monads.
- Composable typed actors.
- It also augments a lot of Scala standard library classes such as List and Option with more functional features. It replaces some of the standard library abstractions with improved variants (for example, it offers a Disjunction type (\vee) that's an implementation of a right-biased Either and is far more useful than the one offered by the standard library).

For more details on Scalaz, see <https://github.com/scalaz/scalaz>.

WHY A GENERIC MODULE FOR APPLICATIVE?

As you must have noticed, you've defined a generic module for applicatives. To get an Applicative for any data type, you need to define a specialization for that data type that will implement the abstract functions that the generic abstraction provides. For example, to make an Applicative for List, you need to define something like the following:

```
def ListApply: Applicative[List] = new Applicative[List] {
  def map[A, B](a: List[A])(f: A => B): List[B] = a map f
  def unit[A](a: => A): List[A] = List(a)
  def ap[A, B](fs: List[A => B])(as: List[A]): List[B] = for {
    a <- as
    f <- fs
  } yield f(a)
}
```

`ListApply` gives you an `Applicative` instance for a `List`, which means that you can implement sequencing of effects in the applicative way on elements of the `List`. This makes `Applicative` an open abstraction, and you can extend any of your abstractions post hoc to make it an `Applicative`. This is a significant difference from extension through subtyping, whereby you need to define a class as a subtype up front when you define that type.

Another advantage that a generic applicative (or a functor) definition offers is the ability to define generic combinators on them. Here's an example from Scalaz:¹⁵

```
def traverse[F[_]: Functor, G[_]:Applicative, A, B](fa: F[A])
  (f: A => G[B])(implicit l: Foldable[F]): G[F[B]] = ...
```

Looking at the type of the function, you realize that it provides a way to do effectful traversals over a sequence that's a `Foldable`.¹⁶ Here's a typical use of this function in our domain model. Say you have a list of accounts, some of which may be closed. You need to write a function that gives you a list of balances of those accounts only if no account is closed. If any account is closed, the function does nothing and returns an appropriate type indicating that.

```
import scalaz._
import Scalaz._
val accounts: List[Account] = //..
Applicative[Option].traverse(accounts) { a =>
  if (a.dateOfClose.isDefined) None else a.balance.some
}
```

Here you use `Option` as an applicative and lift the result of computation within `Applicative[Option]`. As is true for applicatives, all effects are sequenced through, and you get the final list within an `Option` if none of the accounts is closed, and a `None` otherwise. As a domain modeler, you get immense power out of such generic combinators; note that `traverse` is completely parametric over the type parameters and the constraints you define as part of the algebra. And with appropriate type specialization, you can implement domain-specific traversal functionality that you saw just now.

WHAT GOOD IS THE APPLICATIVE FUNCTOR PATTERN FOR YOUR DOMAIN MODEL?

Now that you've seen how the pattern evolved through a specific use case of domain modeling, let's discuss how useful it is from a generic pattern point of view. As you must have realized by now, the central idea behind the pattern is the sequencing of effects. You have a bunch of contexts (effects) evaluated; in our example, these are the validation functions that return an instance of `Validation`. The main idea here is that *all* of these contexts will be evaluated, regardless of the result that each generates. Some of the validations can fail, but this won't prevent others from executing. All

¹⁵ Scalaz defines `traverse` in a separate class for traversable applicatives; here I'm simplifying a bit.

¹⁶ We discussed `Foldable` in section 4.1.2.

contexts are evaluated independently when you map over an applicative functor. Only when all of them are successful does the function create the new context.

So apply the Applicative Functor pattern when you need to execute contexts that are independent of each other in sequence. The shape of the computation is preserved in the sense that all contexts get executed,¹⁷ and execution of one isn't dependent on the success of another context. This differentiates an applicative functor from a monad, which is covered next. Figure 4.2 illustrates an applicative as a shape-preserving sequencing of effects.

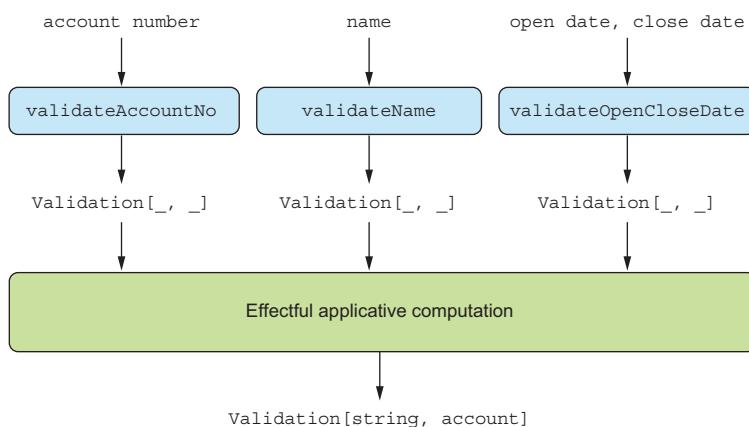


Figure 4.2 Applicative computation preserves the shape of the computation. All validations are computed regardless of whether any of them fails before the computation is threaded through the applicative.

Validation is one use case for which applicative functors make sense. Another example is executing a sequence of independent batch processes and then computing the result. The fact that the processes are executed as *independent* effects gives you leverage to parallelize their execution (maybe using Future-based computation). This is yet another advantage that this pattern offers: You can strategize your evaluation policy of all the contexts beforehand because you know all of them will be executed.

4.2.3 Monadic effects—a variant on the applicative pattern

I'm sure by this time you're familiar with monads. Chapter 3 covered monads in the context of functional combinators, sequencing effects using `flatMap`, and dependency injection of repositories into domain services. But every example referenced specific type constructors (such as `List` or `Option`) implementing `flatMap`. In this

¹⁷ By the shape of the computation being preserved in applicative effects, I mean that all inputs are processed regardless of the outcome of any of them. Contrast this with monadic effects, where one computation can determine whether any other will take place. All inputs may not be computed in a monadic application.

section, you'll learn how to build a generic pattern out of those specializations. As with an applicative, you'll build a `Monad[T]` with the algebra necessary to implement the desired effects. The semantics of the effects will be different from that of an applicative, and you'll see how to use the semantics in a different modeling context than an applicative.

This section describes monads from the following four perspectives:

- How a monadic effect differs from an applicative effect
- A look at the generic monad module with respect to its implementation in Scalaz
- An example of a specific type of monad, the `State` monad (not a specific type constructor implementing a monadic effect) and how you can use it effectively in your domain model
- How you can use the generic monad combinators to implement specific functionalities of your domain model

THE GENERIC MONAD MODULE

What does a generic monad module look like in Scala? A monad implements different semantics than an applicative. But a monad *is also* an applicative, and that makes it a more powerful abstraction than an applicative. Apart from supporting all the functions that an applicative does, a monad also implements `flatMap`:

```
trait Monad[F[_]] extends Applicative[F] {
    def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]
    //.. other Applicative functions
}
```

You've seen `flatMap` as a method in `List`, `Option`, and many other classes in the Scala standard library. But what does the algebra of a generic `flatMap`, as defined in the preceding `Monad` module, tell you? It looks similar to the `map` combinator that you saw in `Functor`. The only difference is the signature of the function, `f`, that a `flatMap` takes. In the case of `map`, the input function returns a value, but in `flatMap` it returns an instance of the type for the monad (for example, `List` or `Option`). When you apply the function `f` to the input `ma` of type `F[A]` in `flatMap`, you end up with `F[F[B]]`. And the `flatMap` joins the two, flattening them into one single effect, `F[B]`. Semantically, a `flatMap` is equivalent to a `map` followed by a `flatten`.¹⁸

The next important point regarding `flatMap` is to understand how the chaining of computations takes place when you plug in multiple `flatMap`s together. They compose sequentially, and the chain breaks if one of the `flatMap`s breaks. This is different from the way that applicatives work. Next you'll see how to use this to implement domain modeling use cases that need to fail fast.

¹⁸ For a complete explanation of how to implement monads, see *Functional Programming in Scala*.

HOW IS A MONADIC EFFECT DIFFERENT FROM AN APPLICATIVE EFFECT?

The previous section indicated how an applicative effect preserves the shape of the computation. Because you know beforehand the steps of your computation, you can perform optimizations, such as parallelizing some of them for better throughput. The situation is a bit different in the case of monadic effects.

Consider the following algebra for the `AccountService` module, which has methods that return a `Try[_]` to take care of the fact that operations may fail:

```
trait AccountService {
    def open(no: String, name: String, openDate: Option[Date],
            r: AccountRepository): Try[Account]
    def close(no: String, closeDate: Option[Date],
              r: AccountRepository): Try[Account]
    def debit(no: String, amount: Amount,
              r: AccountRepository): Try[Account]
    def credit(no: String, amount: Amount,
               r: AccountRepository): Try[Account]
    def balance(no: String, r: AccountRepository): Try[Balance]
}
```

As you know, `Try` is a monad, and you can chain multiple operations from `AccountService` to model some larger domain behaviors. Here's an example that sequences a bunch of credits and debits and inspects the final balance on a customer account:

```
object App extends AccountService {
    val r: AccountRepository = ...
    def op(no: String, r: AccountRepository) = for {
        _ <- credit(no, BigDecimal(100), r)
        _ <- credit(no, BigDecimal(300), r)
        _ <- debit(no, BigDecimal(160), r)
        b <- balance(no, r)
    } yield b
}
```

Assume that you have an instance of `AccountRepository` and you invoke `op` on an existing account number (meaning the account is present in the repository). `op("accountNumberExists", r)` will result in a computation being executed that threads through all the calls of `debit`, `credit`, and `balance` in the implementation of `op`. And as you know, `flatMap` and `map` will be used to sequence this effect through the full chain of computation.

Now consider the other case, where you pass `op` an account number that doesn't exist in the repository. `op("accountNumberNotExists", r)` will result in an immediate failure in the first step. The call to `credit(no, BigDecimal(100), r).flatMap { _ => ... }` returns a `Try[Account]`, which is a `Failure`, and the entire chain of `flatMap`s breaks there. So the subsequent calls to `credit`, `debit`, or `balance` are never executed. Compare this with the applicative effects in figure 4.2, where all the validation functions were executed regardless of the result of each of them. So a monadic effect

results in a computation that's conditional, and the execution of one step depends on the success or failure of the previous step in the chain.¹⁹ Figure 4.3 illustrates this concept using the example just discussed.

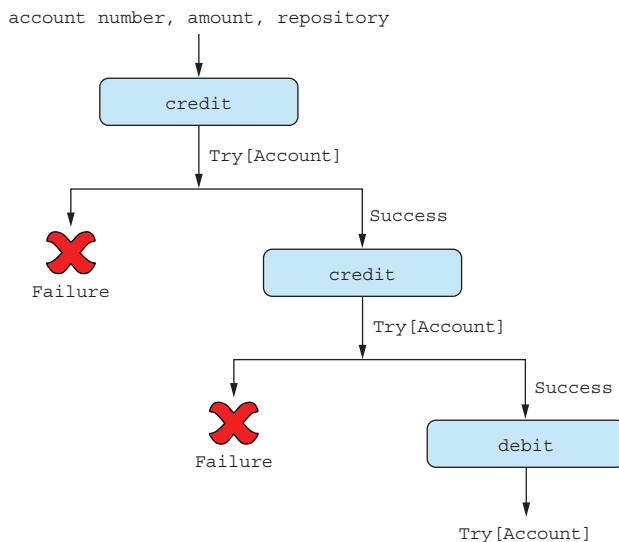


Figure 4.3 In a monadic computation, the shape depends on the result of execution of each step. Here the computation can stop at any of the places marked with an x if the function returns Failure from Try. All the steps of computation are executed only if all previous steps return Success. Compare this with the applicative computation illustrated in figure 4.2.

As you've seen, both applicative and monadic computations have their uses when you model a domain. You need applicative effects when you need to make an independent set of computations, and monadic effects when you have a graph based on conditional execution and need to fail fast.

Monadic and applicative effects in domain-model design

What benefits does the effectful programming model of monads or applicatives add to a domain-model implementation? Both computation structures help you build powerful abstractions so that your core domain model remains clean, concise, and boilerplate free. Here are a few highlights:

- *Power of function application over the additional structure that we call effects—* You've seen how ordinary functions compose and how they can be applied over

¹⁹ To reiterate, the applicative effect executes all functions, independent of the result of execution of any of them.

an argument. But that may not be enough for domain models, where you need to apply a function with custom behavior at the point of application. Look at the function application that `flatMap` offers: `def flatMap[A, B] (ma: F[A]) (f: A => F[B]): F[B]`. It applies `A => F[B]` to `F[A]` and then flattens `F[F[B]]` to `F[B]`. Here you flatten at the point of application so that you get back the structure you supplied to `flatMap`. This enables you to sequence the function application over the effectful structure of `F[_]`.

- *Power of function composition over the additional structure that we call effects*—If you have a function `f: A => F[B]` and another function `g: B => F[C]`, where `F` is a monad, then you can compose them to get `A => F[C]`. This isn’t ordinary function composition; it’s composition of monadic functions. It’s also called *Kleisli composition*; section 4.4 presents a complete example that demonstrates its power. One great example is the `State` monad, which allows you to compose stateful functions and manages threading of the state through them automatically.¹⁹
- *Automatic failure handling*—When you think of sequencing function application or composition, you need to think of handling intermediate failures. Any of the individual functions may fail and need to be propagated to the client of the API. Monads and applicatives handle failure as part of their computation model, and you’ve already seen the differences in the way they do that. But the bottom line is that your core domain logic remains free of boilerplate error checks.

APPLICATIVE OR MONAD: WHICH ONE SHOULD I USE?

Being a subtype of `Applicative`, `Monad` is a more specialized abstraction; it has more power, but applicatives are more general and hence usable in more contexts than monads. While designing your domain model, you’ll also see that although in many cases your initial reaction might be to model using a monad, an applicative can be a better fit. *This is also one way of saying that you should use the least powerful abstraction that works for your use case.* Use applicatives when you can, and fall back to a monad when you need a `flatMap`.

With the `Monad` trait defined as in the preceding code (see Scalaz for a complete implementation of a `Monad`²¹), you can create a monad for any type constructor `F[_]` for which you can implement the algebra in a meaningful way. But you can also create special-purpose monads by using subtyping to extend the `Monad` trait. Because Scala is an object functional language, you can use parametric polymorphism (extension through different instances of `F[_]`) as well as subtype polymorphism (extension through subtyping). The next subsection covers one such specialization—the `State` monad—and then uses the `State` monad to discuss how to use generic monadic combinators to implement behavior of your domain model. As you’ve been doing so far, you won’t implement any combinator; you’ll use the ones from Scalaz and focus more on how to use them for implementing specific behaviors of your model.

²⁰ Section 4.2.3 covers the `State` monad.

²¹ Scalaz’s implementation of `Monad` is a bit different from what’s discussed here. It’s closer to how Haskell defines it. It defines a separate abstraction for `Bind` and then mixes it in with `Applicative` to form a `Monad`. But the basic concepts are the same.

THE STATE MONAD—MANAGING STATEFUL COMPUTATIONS IN YOUR MODEL

I've talked a lot about how functional programming is about passing values as arguments to pure functions. But when you model a domain in real life, frequently you need to manage states, which you don't pass to functions. But you do need to keep track of them, read values from them, write values into them, and treat them as mutable objects. A common example of stateful programming is random-number generation, or more precisely, pseudo-random-number generation. Here you have an initial value, the seed, which is used as the basis to generate a sequence of numbers that have the appearance of being random in nature. The random-number generator does so by maintaining a global state that's updated every time a new number is generated. But when you invoke the random-number generator function, you never pass the state around explicitly. You invoke calls such as `scala.util.Random.nextString(10)`, which generates a random `String` value of length 10 for you.

One way to handle the management of this stateful computation is through explicitly mutable data structures. In Scala, you can use `var` to store a state and modify it directly when you need to change the state. Or in functional programming, you may choose to carry the state as you move along, much as you do in tail recursive calls. Carrying the state as you go along is cumbersome and definitely calls for a better abstraction. The State monad gives you this abstraction that does the threading of the state as part of its computation structure, and it does that using the sequencing power (`flatMap`) of the monad. As a user of the abstraction, you need only to provide the application logic—how to modify the state or how to fetch the value from the computation. And you do this again by passing pure functions. As you've been doing, you'll use the State monad implementation from Scalaz and look at an example from our domain.

Let's consider an example: You've fetched a list of balances for several accounts and would like to update each balance with the transactions that have occurred during the day for all of those accounts. This is a typical operation that's done in the banking world through batch processes. An account balance that the back office maintains may not be updated instantly when the user performs a transaction. The following code defines a few type aliases for easier readability and assumes that the list of balances has been fetched in a `Map` from account numbers to their balances:

```
type AccountNo = String
type Balances = Map[AccountNo, Balance]
val balances: Balances = ...
```

This `Map` is the state that you need to manage. You query the `Map` to get the existing balance and update the `Map` when you process a transaction. Here's what a simple model of a `Transaction` looks like:

List of transactions that happened during the day

Transaction is the amount transacted for the account that day. A negative amount indicates a debit transaction, whereas a positive amount indicates a credit transaction.

```
case class Transaction(accountNo: AccountNo, amount: Money, date: Date)
→ val txns: List[Transaction] = ...
```

You'll use the State monad to manage this stateful computation of the balance. Let's import the State module from Scalaz and the relevant object that gives you the combinators that the monad defines, as shown in the following listing.

Listing 4.5 Balance update using the State monad

```

import Monoid._           ←
import scalaz.State        ←
import State._             ←

def updateBalance(txns: List[Transaction]): State[Balances, Unit] =
  modify { (b: Balances) =>
    txns.foldLeft(b) { (a, txn) =>
      implicitly[Monoid[Balances]].op(a, Map(txn.accountNo ->
        Balance(txn.amount)))
    }
  }

```

**Use your own Monoid implementation that
accompanies this chapter's code. You can use the
Scalaz Monoid to make your code even more succinct.**

**The State module
from Scalaz**

**The object import gives
you the combinators
that you'll use shortly.**

The following list provides the highlights of this code, which uses monad combinators in managing the account balance:

- State is a monad defined as `State[S, A]`, which is nothing but a function from `(S) => (A, S)`, which takes as input a state S and produces an output state S and a value A. In this case, the state is `Map[AccountNo, Balance]`. The value can be any information you want to extract as part of the computation. But you don't want any for this use case; you just want to update the state with the transactions that have occurred throughout the day. So for this use case, you have `State[Map[AccountNo, Balance], Unit]`.
- `modify` is a combinator defined on the State monad as `def modify[S](f: S => S): State[S, Unit]`. It modifies the state and returns the updated state within the monad. You need to pass a pure function to `modify`, which it will use to update the state and return the modified state—exactly what you would have done if you cared to carry the state yourself and thread it across your computation.
- In `updateBalance`, you use an update function that takes the `Map` and modifies the balance corresponding to the input account number. Incidentally, `Balance` is a monoid.²² Hence you can use it as a value in `Map`, which then becomes a monoid. So you can use the power of monoids to make this update look easy and readable—one more example of the benefits of programming to the proper level of abstraction.

²² For a definition of a Monoid for `Balance`, see the code in this chapter's online repository.

What does `updateBalance` return? It returns the `State` monad: `State[Balances, Unit]`. So you haven't yet supplied the monad enough input to *run* the computation; you've just built the computation by composing the abstractions that you need. Nowhere in `updateBalance` have you supplied the initial state, the `Map` named `balances` from where all computations should start. You've just expressed your intention that `modify` needs to be invoked for updating the state and it should take the `Map` as its input. This is one of the reasons that FP is so composable; you compose abstractions in small modules and then glue together the various parts based on your requirements. You have more reusability and better modularization of code.

If you want to run `updateBalance` on a list of transactions, the monad has a `run` method that executes the computation, given an initial value of the state:

```
updateBalance(txns).run balances
```

From the point of domain modeling, what does the `State` monad buy you? Besides giving you all the monadic effects discussed in the previous section, `State` helps you carry your model state, so that you don't have to do it in your application code. It's all about delegating the plumbing logic to the monad itself. And it does this in a completely generic manner so that you can reuse the `State` monad and its combinators for managing *any* kind of state. As a domain modeler, you just have to specify the business logic of how you'd like to change the state as a pure function. The monad uses the power of function composition and supplies you a nicely packaged abstraction dressed up with a set of combinators.

State monads in domain modeling

You need to manage mutable states in domain models; you can't have everything stateless in real models. Use the `State` monad that makes state handling referentially transparent through automatic threading of states across the composed functions. In the example covered in this section, convince yourself that `updateBalance` is referentially transparent and provide an example of how to handle modifications through the `State` monad.

THOSE PESKY MONAD COMBINATORS—WHAT GOOD ARE THEY?

Our discussion of the `State` monad indicated that the combinator `modify` accepts a pure function and uses that to update the state of your model. But `modify` is specific to the `State` monad, and we've been talking about generic combinators that you can use uniformly across all monads. This section presents one of them and discusses its applicability to implementing a specific feature of your model. You can shape your problem to fit the requirements of such combinators. This may be a bit nonintuitive initially, especially if you come from a background of imperative programming. But after you get used to thinking in terms of functions, such combinators will start making more sense.

In our domain model, every account has an account number that has to be unique across the whole system. It's no surprise that the model needs to have logic for generating these numbers. This logic varies across implementations, but each of them tries to generate a unique number, taking into consideration several factors. But even after a number is generated, you need to check whether it has already been allocated to another account—just an obligatory check, because algorithms fail, numbers overflow, and lots of strange things happen in computation. In summary, you need to generate and check until you come up with a unique number for an account. Let's have an abstraction for such a generator that generates an account number and also has a function to check for uniqueness:

```
final class Generator(rep: AccountRepository) {
    val no: String = scala.util.Random.nextString(10)
    def exists: Boolean = rep.query(no) match {
        case Success(Some(a)) => true
        case _ => false
    }
}
```

Queries the repository to check for uniqueness

AccountRepository is the repository, which stores accounts. See chapter 3 for details on Repository.

The generation logic will be complex. Assume random strings for the time being.

If you think in terms of the State monad, this Generator class is the state of your computation. It generates an account number and allows you to check for its uniqueness. If the number exists for another account, you can create another instance of Generator to get another number for your account. The following code that uses a mutation is the imperative version of this logic. The state gen is being mutated until you have the proper instance that gives you a unique number:

```
var gen = new Generator(r)
while (gen.exists) {
    gen = new Generator(r)
}
```

You can use Generator as the state and abstract the preceding imperative logic into a nice combinator that has a pure functional interface to the user. In Scalaz, this combinator is called `whileM_` and is defined as follows:

```
def whileM_[A] (p: F[Boolean], body: => F[A]): F[Unit] = ..
```

The action, `body`, is executed repeatedly as long as the Boolean expression returns `true`. You need only the final state from where you can fetch the account number generated, so you can discard the result. The combinator `whileM_` uses the power of pure functions to transform this imperative logic into a declarative form. Modeling this imperative logic by using this combinator is straightforward and is presented in the following listing.

Listing 4.6 Using a monadic combinator to generate valid account numbers

```
val StateGen = StateT.stateMonad[Generator]
import StateGen._
val r: AccountRepository = //..
val s = whileM_(gets(_.exists), modify(_ => new Generator(r)))
val start = new Generator(r)
s exec start
```

With the knowledge of the State monad still fresh in your mind, the combinator should look intuitive enough. You'll iron out the details soon. But the most important point is that *you're still dealing with pure functions and their compositions*. Here are the rest of the facts that explain how `whileM_` does what you expect it to do:

- `StateT` is a monad transformer. It's not important what it does here, except that the `stateMonad` function on `StateT` generates a State monad, with `Generator` as the state.
- The `import` in the second line makes available the various combinators within the State monad as well as the generic Monad abstraction. (Remember, `State` is also a monad.)
- `modify` is the combinator you used earlier that takes a function to modify the state. In our case, it's creating another instance of `Generator` that will generate another account number.
- `gets` applies the transformation function passed to it to the value and returns the updated State monad. So here it checks whether the generated account number already exists. As long as `exists` returns `true`, you need to execute the body of `whileM_`.
- If you want to execute the computation and generate the next valid account number, you need to pass a start configuration of `Generator` and fire an `exec` on the monad.

`whileM_` is a combinator that works for all monads, and the preceding discussion is an example of how such generic abstractions make great building blocks for your domain model. Identify the use case that fits the combinator and supply the functions that it needs. And the combinator will do the whole plumbing behind the scenes for you.

4.3 How patterns shape your domain model

You've seen quite a few patterns by now. Besides the basic techniques of function composition, you saw patterns such as monoids that help in composing algebraic structures over an associative binary operation. In this chapter, you've seen patterns of effectful programming—two types of effects that shape the structure of your computation. You've seen examples of how they can be used in the context of your domain model. But how useful are these patterns? Are they something that you can use rarely and in the context of few use cases? Or are they used frequently enough to be discussed at length in a book on domain modeling?

Previous chapters reiterated that the essence of functional programming lies in the power of pure functions. Add static types to the mix, and you have algebraic abstractions—functions operating on types and honoring certain laws. Make the functions generic on types and you have parametricity. The function becomes polymorphic, which implies more reusability, and if you’re disciplined enough not to leak any implementation details by sneaking in specialized types (or unmanaged hazards such as exceptions), you get free theorems.²³ This is what we call the *ladder of goodness* as far as functional abstractions or patterns go. Figure 4.4 displays this ladder of goodness, which you can progress incrementally as you wrap your head around the various nuances of functional programming.

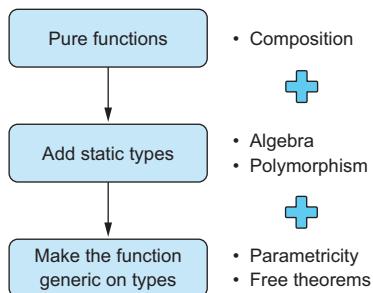


Figure 4.4 The ladder of goodness. When you have pure functions, you can compose fearlessly. Add static types, and you get algebraic abstractions. And when these functions are completely generic on the type parameters, you get free theorems.

PATTERN GOODNESS IN DOMAIN MODELING #1 Patterns are mostly useful when they’re generic and parametric over types. In such cases, you can build structures that can be implemented for any type that complies with the algebra of the abstraction. When you have a generic module for `Monad[F[_]]`, you can have specific implementations for any type constructor (for example, `List` and `Option`).

By virtue of being generic, a pattern gives you the scope of defining generic combinators, which you saw in the cases of monads and applicatives. These combinators are implemented once and for all and apply to every specialization that you implement for that pattern. You saw examples of this with the `whileM_` combinator defined in the `Monad` module and the `modify` or `gets` combinators defined for the `State` monad. Don Stewart describes this approach: “Name a concept once. Implement it once. Reuse the code forever.”²⁴

PATTERN GOODNESS IN DOMAIN MODELING #2 Being generic in nature, patterns offer generic combinators and functions that are applicable for all

²³ When you have a polymorphic function, you can get some theorems automatically based on the types of the function. These are some free theorems that parametricity buys us. Phil Wadler’s paper “Theorems for Free,” available from <http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html>, has more details.

²⁴ See “Haskell in the Large” by Don Stewart (<http://web.archive.org/web/20150129012424/http://code.haskell.org/~dons/talks/dons-google-2015-01-27.pdf>).

specific instances of the patterns. The combinator sequence²⁵ defined on an Applicative module can be used for any instance of Applicative (for example, Applicative[List] and Applicative[Option]).

Let's now take a deeper look at the architectural value that these patterns add to your domain model. Consider the applicative effect that you implemented to validate account attributes before creating an account in section 4.1.2 (figure 4.2 illustrates this concept). In the absence of the Applicative Functor pattern, the code would be much more verbose and the application code would have to handle lots of incidental complexities such as error handling, chaining of validation for all the attributes, and conditional execution based on the outcomes. This would have made our application code much more complex, and real domain logic would have been cluttered within these accessory concerns. Figure 4.5 clearly illustrates the concerns that the generic applicative functor abstraction handles and the ones that your domain logic handles.

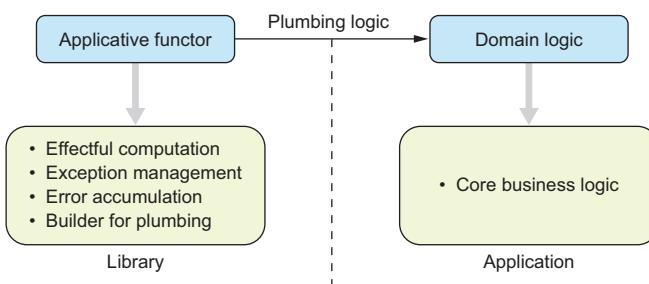


Figure 4.5 Patterns are generic structures that can be glued onto the domain model by using plumbing code. Note the number of concerns that a good design pattern can handle on its own. This takes a lot out of the application code, which becomes more succinct and expressive in the absence of incidental complexities.

PATTERN GOODNESS IN DOMAIN MODELING #3 Patterns abstract away the commonality of abstractions and leave the user to implement only the domain-specific variability. This makes your application code concise, succinct, and more expressive without any incidental complexity.

Design patterns compose—at least, they should. You should be able to compose multiple patterns and come up with larger ones. Let's consider the Validation abstraction from Scalaz. It's similar in usage to what we discussed in section 4.1.2, but has a wide range of combinators, interoperability with other abstractions, and a lot of things that make it a production-ready structure to address validation of domain objects. Here's the basic structure of Validation in Scalaz:

²⁵ sequence is defined in the type class `Traverse` in Scalaz at <https://github.com/scalaz/scalaz/blob/series/7.2.x/core/src/main/scala/scalaz/Traverse.scala>.

```
sealed abstract class Validation[+E, +A] { ... }
final case class Success[A](a: A) extends Validation[Nothing, A]
final case class Failure[E](e: E) extends Validation[E, Nothing]
```

This looks awfully similar to `scala.Either[+A, +B]`, which also has two variants in `Left` and `Right`. In fact, `scalaz.Validation` is isomorphic to `scala.Either`.²⁶ In that case, why have `scalaz.Validation` as a separate abstraction? `Validation` gives you the power to accumulate failures, which is a common requirement when designing domain models. A typical use case arises when you're validating a web-based form containing many fields and you want to report all errors to the user at once. If you construct a `Validation` with a `Failure` type that has a `Semigroup`,²⁷ the library provides an applicative functor for `Validation`, which can accumulate all errors that can come up in the course of your computation. This also highlights an important motivation for using libraries such as Scalaz: You get to enjoy more powerful functional abstractions on top of what the Scala standard library offers. `Validation` is one of these functional patterns that make your code more powerful, succinct, and free of boilerplates.

In our discussion of applicative functors and the use case of validation of account attributes, we didn't talk about strategies of handling failure. But because in an applicative effect you get to execute all the validation functions independently, regardless of the outcome of any one of them, a useful strategy for error reporting is one that accumulates all errors and reports them at once to the user. The question is, should the error-handling strategy be part of the application code or can you abstract this in the pattern itself? The advantage of abstracting the error-handling strategy as part of the pattern is increased reusability and less boilerplate in application code, which are areas where FP shines. And as I've said, a beautiful combination of Applicative Functor and Semigroup patterns enables you to have this concern abstracted within the library itself. When you start using this approach, you'll end up with a library of fundamental patterns for composing code *generically*. And types will play a big role in ensuring that the abstractions are *correct by construction*, and implementation details don't leak into application-specific code. You'll explore more of this in exercises 4.2 and 4.3 and in the other examples in the online code repository for this chapter.



EXERCISE 4.2 ACCUMULATING VALIDATION ERRORS (APPLICATIVELY)

Section 4.2.2 presented the Applicative Functor pattern and used it to model validations of domain entities. Consider the following function that takes a bunch of parameters and returns a fully-constructed, valid `Account` to the user:

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): ValidationNel[String, Account] = { ... }
```

²⁶ Intuitively, an isomorphism is a slightly more relaxed relationship than equality. Whereas equality mandates that the two abstractions have to be exactly equal, isomorphism says that the two abstractions have to be convertible both ways. In the current case, given `Validation[E, A]`, you can construct an equivalent `Either[E, A]`, and vice versa.

²⁷ A semigroup is a monoid without the zero.

- The return type of the function is `scalaz.ValidationNel[String, Account]`, which is a shorthand for `Validation[NonEmptyList[String], Account]`. If all validations succeed, the function returns `Success[Account]`, or else it must return *all* the validation errors in `Failure`. This implies that all validation functions need to run, regardless of the outcome of each of them. This is the applicative effect.
- You need to implement the following validation rules: (1) account numbers must have a minimum length of 10 characters, (2) the rate of interest has to be positive, and (3) the open date (default to today if not specified) must be before the close date.
- Hint: Take a deep look at Scalaz's implementation of `Validation[E, A]`. Note how it provides an `Applicative` instance that supports accumulation of error messages through `Semigroup[E]`. Note `Semigroup` is `Monoid` without a zero.
- Hint: A good place to start the implementation is from the definition of the same function in the discussion on smart constructors in section 3.3.2 and its implementation in the online code repository.

PATTERN GOODNESS IN DOMAIN MODELING #4 Patterns compose. You can combine multiple smaller patterns to come up with larger patterns.

Now that you have an idea of how using design patterns can improve the quality of your domain model by helping you build better abstractions that you can reuse across models, let's take a look at a couple of field stories. These are use cases from domain models built using functional programming that use a combination of types, algebra, and patterns toward better abstraction, modularity, and maintainability. Section 4.4 presents a complete case of API evolution that starts with the algebra and incrementally builds upon the abstractions using the power of types, patterns, and function composition. You'll see how much you can do with these artifacts before you commit to any specific implementation.



EXERCISE 4.3 FAIL-FAST VALIDATION (MONADIC)

Section 4.3.3 covered monadic effects and how they compare with applicative ones. In exercise 4.2, you explored the applicative way of handling validations. In this exercise, you'll explore the monadic application of effects. Let's consider the same function as exercise 4.2, but implement fail-fast validations of the account attributes:

```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): ??? = { //..
```

- a The return type of the function is something that you need to decide. The idea is to return a monad, which when executed will result in either a successfully constructed `Account` or a failure containing the first validation information that failed. Hint: Scalaz has a nice abstraction for this.

- b You need to implement the following validation rules: (1) account numbers must have a minimum length of 10 characters, (2) the rate of interest has to be positive, and (3) the open date (default to today if not specified) must be before the close date.
- c Implement every validation as a separate function and combine them using a for-comprehension within the main function, `savingsAccount`.

4.4 Evolution of an API with algebra, types, and patterns

Chapter 3 showed how to evolve an API starting with the algebra of the intended functionality of the domain. In this section, you'll look at a more interesting example that uses the same algebraic approach to discover how existing patterns of FP fit nicely into composing domain behaviors leading to supple API designs. You'll start with a typical use case, think of the algebra for supporting the use case, pull types out of thin air, and then try to make them compose. It will be an interesting exercise, and the end will be illuminating enough to convince you of the power of function composition through usage of existing patterns. But first you need to set up some enhancements of the domain that we've been discussing. You'll step out of the personal banking domain and increase the scope to areas of investment banking. Instead of talking about debits and credits, you'll talk about client orders, execution, trade, and settlement. Typically, many financial institutions that offer personal banking also have a separate business of investment banking that serves institutional and retail clients. The following sidebar presents some basic concepts in investment banking that you'll use in this example.

Basics of security business: order, trade, execution

The specific area of investment banking for this example relates to the business of trading and settling securities on the exchange (also known as the stock market). The following are some basic concepts that you need to know in order to follow the example:

- Just as you open an account with a bank for personal banking, you as a client can open a trading or a settlement account with an investment bank. If you open a trading account, you can trade securities (stocks and bonds) through it. A trade can be either a purchase or sale of securities. A settlement account is one through which your trades get settled.
- If you're a client with a trading account, you can place orders for trade with the bank. An order consists of the name (or ISIN) of the security to trade (buy or sell), quantity to be traded, price range, and other information that isn't essential for our current context.
- After the orders from clients reach the trading organization (which is the investment bank), they're placed on the exchange (market) for execution.

(continued)

- The market does the trading and sends back *executions* to the bank. These aren't mapped to the specific orders that each individual client has placed. The executions are matched against the bulk of all orders that all clients have placed on that particular day.
- After the bank receives the executions, it performs the process of *allocation*, which maps executions to individual accounts in the form of *trades*. These trades have a mapping with the order that the client had placed with the bank.

The use case for our example goes through this sequence; hence it's not essential to get into further details. If you're interested, Investopedia (<http://investopedia.com>) provides more details on investment banking.

Here's the use case for our example:

- The client places order in their own specific format to the trader.
- The trader transforms it to an internal format (a domain entity for the client order) and places the order in the market.
- When trading is done in the market, the trading agency gets back what we call executions. Note that the execution isn't specific to one client order. All client orders are aggregated and then placed for trade in the market. So you aggregate a bunch of client orders and place that basket in the market for execution.
- After the trading agency gets back the executions, they're allocated to client accounts as trades and the respective client balances are updated.

Without considering any implementation details, let's assume some types for the domain entities and value objects and try to create a first draft showing how these functions will look.

4.4.1 The algebra—first draft

The first rule of domain API design is to stick to the ubiquitous language of the domain. Let's do that and get to the first level of such an API design, as shown in the following listing.

Listing 4.7 Algebra for the Trading API

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute: Market => Account => Order => List[Execution]
  def allocate: List[Account] => Execution => List[Trade]
}
```

So you have these functions in a module that's named based on the behavior that it models. You've just pulled some types out of thin air with no idea about their implementation. Hence our module, `Trading`, is parameterized on those types. And you aren't

supposed to talk about the implementation of any of them when you're specifying the algebra—that's for the interpretation to take care of. But looking at the preceding specification and with the domain knowledge that you have, you can identify Market, Order, Execution, Trade, Account, and ClientOrder as examples of domain entities. The type signatures of the APIs also subsume some of the rules of the domain (for example, one ClientOrder can lead to multiple instances of Order).

If you squint a bit hard at these three functions, you'll realize these are the exact steps that you need to take in order to serve the use case described in section 4.4. Figure 4.6 illustrates the steps of the use case, using the three functions for which you defined the preceding algebra.

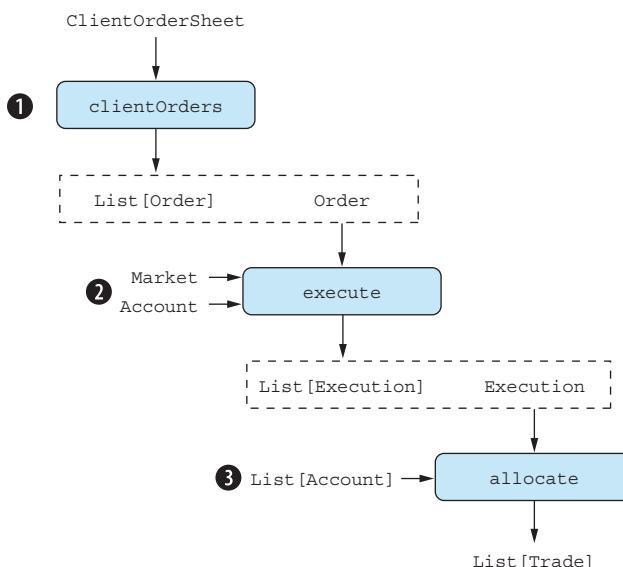


Figure 4.6 The Trading use case steps modeled through the functions defined in the text. The diagram shows the inputs and outputs of the functions—see the text for details on how to compose them together for a sequential execution of the use case.

4.4.2 Refining the algebra

If you look at the algebra of the functions defined in listing 4.7, it's not entirely obvious how to wire them to get the desired sequencing that figure 4.6 requires. The dotted rectangles in the figure give a hint. `clientOrders` outputs a `List[Order]`, and `execute` needs an `Order`. Let's reorganize the function arguments a bit more and have a second look at the result in the next listing.

Listing 4.8 Algebra for the Trading API (slightly refactored)

```

trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute(m: Market, a: Account): Order => List[Execution]
  def allocate(as: List[Account]): Execution => List[Trade]
}
  
```

Generalizing this algebra, you have three functions with signatures $A \Rightarrow M[B]$, $B \Rightarrow M[C]$, and $C \Rightarrow M[D]$ and your intention is to compose them together. And do you know anything about M ? In the current example, M is a `List`, and `List` is an *effect*, as you saw earlier (in chapter 2). But in functional programming, you always try to generalize your computation structure so that you can get some patterns that can be reused in a broader context. It so happens that you can generalize the composition of the three functions over a more generalized effect, a `Monad` instead of a `List`. This is achieved through an algebraic structure called a *Kleisli arrow*²⁸ that allows functions $f: A \Rightarrow M[B]$ and $g: B \Rightarrow M[C]$ to compose and yield $A \Rightarrow M[C]$, where M is a `Monad`. It's just like ordinary composition but over effectful functions.

`Kleisli` is just a wrapper over the function $A \Rightarrow M[B]$. You can compose two `Kleisli`s if the effect (type constructor) is a monad. Here's part of the definition for `Kleisli` from `Scalaz` (simplified):

```
case class Kleisli[M[_], A, B](run: A => M[B]) {
    def andThen[C](k: Kleisli[M, B, C])(implicit b: Monad[M]) =
        Kleisli[M, A, C]((a: A) => b.bind(this(a))(k.run))
    def compose[C](k: Kleisli[M, C, A])(implicit b: Monad[M]) =
        k.andThen(this)
    //...
}
```

Kleisli just wraps the effectful function. As discussed in section 3.2.1, a computation builds an abstraction without evaluation. Kleisli is such a computation that doesn't evaluate anything until you invoke the underlying function `run`. This way, using `Kleisli`, you can build up multiple levels of composition without premature evaluation. Look at the way you use this feature to build up your trade generation API in this section.

Using `Kleisli` composition, composing our three functions becomes just function composition with effects. This also demonstrates how to express your domain algebra in terms of an existing algebraic structure. This is an extremely important concept to understand when talking about algebraic API design. The following listing shows how the domain algebra gets refined with the new pattern that you've just discovered.

Listing 4.9 Algebra for the Trading API (using the Kleisli pattern)

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
    def clientOrders: ClientOrder => List[Order]
    def execute(m: Market, a: Account): Order => List[Execution]
    def allocate(as: List[Account]): Execution => List[Trade]
}
```

Before the Kleisli pattern

²⁸ The term *arrow* comes from category theory; it maps to a function in Scala or Haskell.

```

trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: Kleisli[List, ClientOrder, Order]
  def execute(m: Market, a: Account): Kleisli[List, Order, Execution]
  def allocate(as: List[Account]): Kleisli[List, Execution, Trade]
}

```



**Using the
Kleisli pattern**

4.4.3 Final composition—follow the types

Now you can use the three functions to build a larger domain behavior of trade generation. Just follow the types and you have the complete trade-generation process (simplified for demonstration), from client orders to the allocation of trades to client accounts.

Listing 4.10 Trade generation from client orders

```

def tradeGeneration(
  market: Market,
  broker: Account,
  clientAccounts: List[Account]
) = {
  clientOrders andThen
    execute(market, broker) andThen
      allocate(clientAccounts)
}

```

Take a good look at the implementation of this domain behavior that generates trade from client orders, and compare it with the business specification laid out in section 4.4. You'll notice that the implementation follows the specification closely, and you get the ubiquitous language for free. This exercise once again demonstrates that you can compose functions at a lower level and patterns at a higher level and come up with an implementation of your domain model that's quite expressive.

In *Domain-Driven Design*, Eric Evans mentions the following qualities of a good domain model:

- Intention-revealing interfaces drawn from the ubiquitous language
- Making implicit concepts explicit
- Side-effect-free functions
- Declarative design

You can see all of these qualities illustrated in our approach using statically-typed functional design patterns as the foundation of our model implementation.



EXERCISE 4.4 DEMYSTIFYING KLEISLI

Listing 4.10 models the complete trade-generation process starting from the client order. It uses the power of the Kleisli algebra to compose the three effectful functions. Kleisli gives you a computation (we discussed this along

with the code in listing 4.9) for which you need to invoke the underlying function for evaluation.

- a Run the preceding `tradeGeneration` function for a market, a broker account, and a list of client accounts. What are the types and the results of this computation? You can get the supporting abstractions (market, account, and so forth) from the online code repository for the book.
- b In step (a), do you get the final list of trades as the output? Convince yourself of the result that you get as the output.
- c What would you do additionally to generate the final output of the list of trades?

4.5 **Tighten up domain invariants with patterns and types**

When you model a domain, the behaviors need to honor the constraints of the business. There are many ways to enforce these constraints. You can write pure business logic and enforce them in runtime behaviors, and this is the most commonly used technique when you are using dynamically typed languages. If you have a static type system, you have many additional options at your disposal. Types can be used to great effect in encoding many of the domain constraints. And when you can do this, you get a lot of things for free. You get free tests; the compiler does them for you. And you get parametricity as well, which is much more than writing tests in any form. This section presents another combination of functions and types in the form of patterns that can be used to encode domain invariants for your domain model.

4.5.1 **A model for loan processing**

Let's consider that our banking model needs to process loan applications from its customers. The bank needs to validate the application, get it approved by the appropriate authority, process the loan if found suitable, and then finally disburse the loan. There's a definite workflow here, and the workflow has a sequence that needs to be honored by your model during the processing of a loan application.

Let's consider the model for a loan application in the following listing.

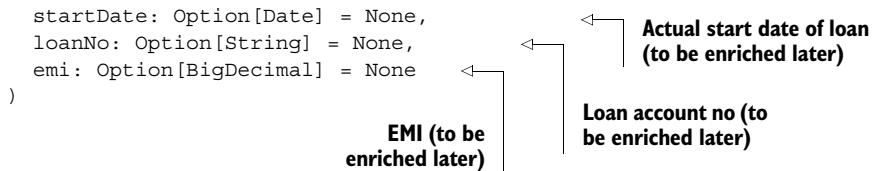
Listing 4.11 The model for a loan application

**Make the constructor private to package;
you'll have alternate means of instantiation.**

```
case class LoanApplication private[Loans] (
    date: Date,
    name: String,
    purpose: String,
    repayIn: Int,
    actualRepaymentYears: Option[Int] = None,
```

The diagram shows the `LoanApplication` constructor with five parameters. Annotations with arrows point to each parameter:

- `date`: Labeled "Date of application of loan".
- `name`: Labeled "Customer's wish for tenure".
- `purpose`: Labeled "Actual tenure approved (to be enriched later)".
- `repayIn`: Labeled "Actual tenure approved (to be enriched later)".
- `actualRepaymentYears`: Labeled "Actual tenure approved (to be enriched later)".



The workflow includes a few distinct steps, and you'll consider the following three for illustration purposes:

- Accept a loan application (from the customer)
- Approve a loan application (by banking authority)
- Enrich a loan application on approval, which fills out the unfilled details of the preceding model

Accordingly, the following contracts model the workflow. The preceding three steps need to be sequenced. And as you learned in section 4.4, Kleisli is a great way to model the sequencing of an effectful function application. The following listing shows the first draft of modeling the workflow.

Listing 4.12 First draft of workflow functions for loan processing

```

def applyLoan(name: String, purpose: String, repayIn: Int,
             date: Date): LoanApplication

def approve: Kleisli[Option, LoanApplication, LoanApplication]

def enrich: Kleisli[Option, LoanApplication, LoanApplication]

```

The `applyLoan` function takes the relevant arguments and constructs an instance of `LoanApplication`. Then `approve` performs the approval process: It returns an `Option` type because there's a chance that the loan application may be rejected. If the loan application is approved, the `approve` function fills out the `loanNo`, `actualRepaymentYears`, and `startDate` fields of the model. `enrich` computes the `EMI` and completes the model. Here's how to use this API:

```

val l = applyLoan("John B Rich", "House Building", 10, today)
val op = approve andThen enrich
op run l

```

The `applyLoan` function is the smart constructor that returns an instance of the model. And the following step uses the Kleisli pattern to wire up the workflow functions in proper sequence to process the loan application.

That was neat! You have the functionality right there wrapped up in all the niceties of a Kleisli. When you design an API, one of the goals is to make it as robust as possible. Or you should make it safe enough so that it's not easy to use it incorrectly without the compiler getting in your way. Consider what happens if in the preceding usage of the API, you write `enrich andThen approve`. The types align for the Kleislis, and the

compiler is happy. But the invariant for the domain behavior is violated: You can't enrich a loan application without approving it. Your runtime behavior is incorrect; the domain invariant violation has bypassed your compiler.

You can't address every violation with the type system. But this is one case where you can use the type system to good effect. The idea is to enforce the sequence of the workflow through more types. Enriching before approval is an illegal state of the workflow, and you need to prevent it.

4.5.2 Making illegal states unrepresentable²⁹

When you have a powerful type system at your disposal, don't be afraid to use more types. And that's exactly what you'll do in this case. You'll use additional types to make your workflow states more explicit. But these types will only play the role of markers; they're there to make a state unique but have no role in the domain logic per se. Let's represent each state of the workflow with a separate type and have the model `LoanApplication` parameterized on this type. The following listing shows the modified model.

Listing 4.13 The loan-processing workflow with phantom types

```
case class LoanApplication[Status] private [Loans] ( //... ←
  trait Applied
  trait Approved
  trait Enriched ←
  type LoanApplied = LoanApplication[Applied]
  type LoanApproved = LoanApplication[Approved]
  type LoanEnriched = LoanApplication[Enriched] ←
  def applyLoan(name: String, purpose: String, repayIn: Int,
    date: Date = today) =
    LoanApplication[Applied](date, name, purpose, repayIn)
  def approve = Kleisli[Option, LoanApplied, LoanApproved] { l =>
    l.copy(
      loanNo = scala.util.Random.nextString(10).some,
      actualRepaymentYears = 15.some,
      startDate = today.some
    ).some.map(identity[LoanApproved])
  }
  def enrich = Kleisli[Option, LoanApproved, LoanEnriched] { l =>
    val x = for {
      y <- l.actualRepaymentYears
      s <- l.startDate
    } yield (y, s)
  }
```

²⁹ The idea of this pattern and the title of this section is from an excellent blog post by Yaron Minsky, which discusses phantom types in OCaml (<https://blogs.janestreet.com/effective-ml-revisited/>).

```

    l.copy(emi = x.map { case (y, s) =>
      calculateEMI(y, s) }).some.map(identity[LoanEnriched])
  }

private def calculateEMI(tenure: Int, startDate: Date): BigDecimal = ...

```

This listing introduces the additional types `Applied`, `Approved`, and `Enriched` that don't participate in any domain behavior implementation. They exist to disambiguate states of the workflow. They enforce the following constraints:

- A new `LoanApplication` is always in an `Applied` state.
- The only function you can invoke on a `LoanApplication` in an `Applied` state is `approve`, and it takes the `LoanApplication` to an `Approved` state.
- You can enrich a `LoanApplication` only in the `Approved` state and take it to the `Enriched` state.

And you do all of these statically and with enforcement from the compiler. You can only `approve` and `then enrich`. The incorrect sequence will make the compiler cry. Because these types are used only for state disambiguation, they're called *phantom types*. And it looks like you've been able to combine phantom types with your pure functional implementation of the API that uses the Kleisli pattern to implement an API that enforces all the domain invariants statically.

The Phantom Type pattern can be effectively used to make illegal states of a domain model unrepresentable. There's nothing functional about this pattern. The reason we discussed this is that it also plays nicely with your functional patterns to add to the robustness of your model.

4.6 Summary

This chapter has been an assembly of advanced topics in typed functional programming. Feel free to read it again to get a full understanding of the topics discussed, and have a look at the code in the online repository. This chapter covered only a few of the patterns; plenty of others are available that you can take advantage of to refine your domain model. The major takeaways from this chapter are as follows:

- *What are functional design patterns?* Design patterns as viewed through the lens of functional programming are different from those in object-oriented programming. In FP, the patterns are the algebra of abstractions, for which you can write specific interpreters depending on your context.
- *The ubiquitous monoid*—Monoid is a design pattern that appears often in domain models. Monoids can make your model generic and help you abstract over the operations of your domain.
- *Patterns for effectful programming*—The three most commonly used patterns in typed functional programming are functors, applicatives, and monads. You can use them to structure the computation in your domain models.

- *Pattern goodness*—You saw a general overview of pattern goodness and how a careful application of functional patterns makes your domain model compositional, parametric, and reusable.
- *Implementation from the field #1*—You learned how to evolve an API by using the power of types and functions. You saw how to use effectful function composition with Kleislis to design a workflow of domain behaviors.
- *Implementation from the field #2*—You saw how to use phantom types to enforce domain invariants statically. You had a lesson on the power of the type system and how to use it to your advantage to encode logic that gets tested automatically by the compiler.

Modularization of domain models

This chapter covers

- Modularizing your domain models
- Exploring a detailed case study of a domain model split into modules
- Understanding how modules aggregate to bounded contexts
- Using an advanced pattern of modularization, the free monads

Whenever you think of a large model, it's always helpful to think in terms of smaller units within the whole. That's how our cognitive abilities work as well. Instead of trying to understand up front how all the details of an entire banking system work, it's easier to think about the smaller parts, such as customer account management, portfolio management, reporting services, and the back-office model. These are also fairly independent subsystems speaking different ubiquitous languages and may well have separate data and domain models. Even within each of these subsystems, there are functionalities that are too complex to understand as a whole.

When I talk about *modularization*, I mean decomposing such a model into smaller modules that group semantically related behaviors. Each module has algebra that's published to clients and one or more implementations that are carefully protected from inadvertent coupling with the client code. This chapter explores the modularization of domain models and how to use abstractions to split your model into composable modules.

Figure 5.1 shows the flow of this chapter's topics. This schematic will guide you to selectively choose your topics as you sail through the chapter.

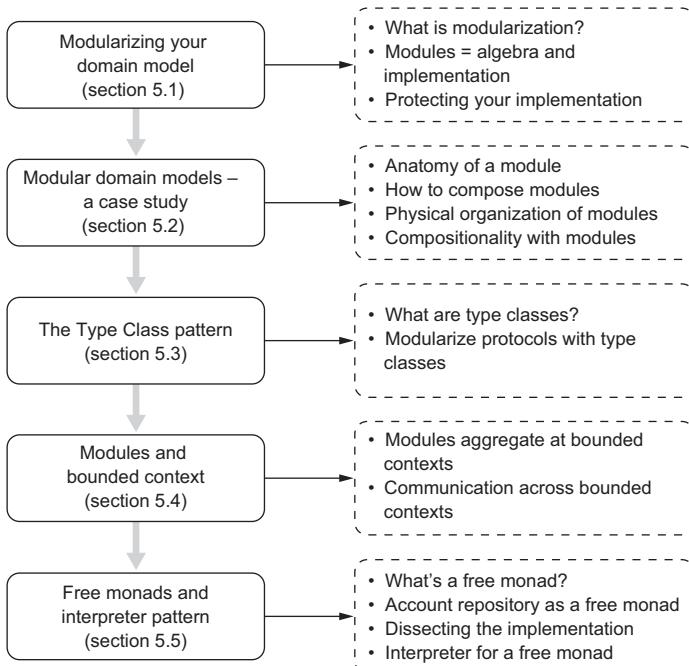


Figure 5.1 The progression of sections in this chapter

At the end of the chapter, you should understand how to use proper abstractions to modularize your domain model.

5.1 **Modularizing your domain model**

When you have a complex system to understand, it's always difficult to comprehend if the system is a big ball of mud designed in a monolithic way. We as human beings always try to find a bunch of simpler parts within the whole that are composed together to form the complete system. This is the way our minds work and the way our cognitive abilities map to things in real life. This process of decomposing a whole system into a collection of simpler parts is the essence of modularization. And the simpler parts themselves are called the *modules*.

After you identify the modules within the system, you expect each module to deliver a specific functionality. In the example of our personal banking domain, you may have a module named Account Service that models all functions needed to manage customer accounts. To deliver this functionality, all elements that are part of this module need to work synergistically among themselves with minimal dependency on other modules. This makes a module strongly cohesive within itself but minimally coupled to other modules. In addition to easing understanding, modularization is of prime importance from a software engineering point of view. A well-modularized system is easier to understand, manage, refactor, and test and gives you more peace of mind in the entire software development lifecycle than a monolithic piece of code base.

What are some of the qualities that a well-modularized implementation of a domain model should have? Let's have a look:

- *Specific and well defined*—Each module delivers a specific functionality, which maps closely to similar behaviors in the problem domain.
- *Cohesive*—A module is strongly cohesive and has loose coupling with other modules.
- *Published contract*—A module never exposes its implementation details and publishes only well-formed algebra to its clients.
- *First class*—A module is implemented in terms of a first-class construct of the language and can be composed to form larger modules.¹
- *Vocabulary*—A module must have a proper name, and all module behaviors must have names—all of which form part of the ubiquitous language of the domain model.

In this chapter, you'll explore some of the patterns of modularization of your Scala-based implementation of domain models. You'll consider a sample case study implementing part of a model from our personal banking domain. This will give you an idea of the following:

- How to think in terms of modules when you have a domain model to implement
- How to define and implement modules using the paradigm of functional programming
- How to compose modules given the fact that Scala supports a first-class module system
- How to organize your code base of entities, value objects, services, and repositories to have a well-organized modularization of the model

¹ Not all languages offer support for first-class modules. Those that do have definite advantages over those that don't. Scala offers first-class support for modules.

5.2 Modular domain models—a case study

Let's consider the following small subset of functionality from the domain of personal banking to illustrate the various aspects of model modularization:

- A few domain elements, entities, and value objects
 - *Account*, an entity that models a customer account.
 - *Balance*, a value object that models a customer balance on an account. Ideally, a balance should have an amount part and a currency part. For simplification, you'll consider the amount part only.
 - *Amount*, a value object that models the amount portion of a balance.
- A few domain services that model business use cases
 - *Account Service*, which implements all functionalities to deal with a customer account. It will be similar to what we discussed in chapters 3 and 4 when you implemented `AccountService` as a domain service.
 - *Interest Calculation*, which implements the logic (simplified) to compute interest on customer accounts. Ideally, it should involve complex computation logic, which is elided here for obvious reasons.
 - *Tax Calculation*, which computes tax on interest accrued on balances in customer accounts.

You'll compose the preceding domain services to implement some meaty domain behaviors. The idea is to illustrate the compositionality that a well-designed modular model has.

For all of these elements, you'll use simplified implementation logic. The main idea of this exercise is to discuss the virtues of good modularization and demonstrate how a well-modularized implementation makes your code base easier to understand, maintain, and refactor.

As you may have noticed, the proposed model already identified a few of the domain behaviors and listed them as separate functionalities (services). I've started talking about smaller chunks of functionalities, and this has made your understanding easier. We haven't yet talked about the final modules that you'll see within this model. But you have the base for our discussion that will help identify the modules as you move on.

Figure 5.2 shows the overall scope of the model that you'll implement incrementally in subsequent sections.

5.2.1 Anatomy of a module

Each domain service in the preceding list forms a coherent set of functionalities of the domain. Each offers to the client the complete suite of functions that are necessary to implement one specific use case. You'll make each of them a separate module because they satisfy the criteria of well-behaved modules laid out in section 5.1.

In this section, you'll take a deep look inside the meatiest module. You'll explore all parts of the implementation that make it organized as a module and see how they

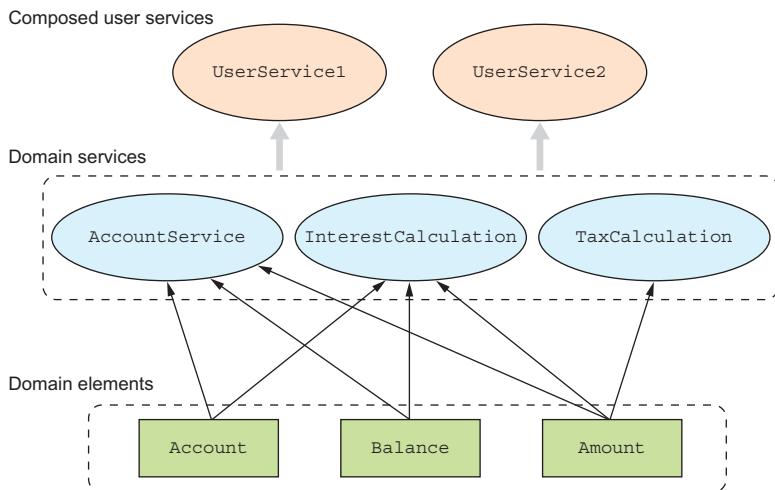


Figure 5.2 Scope of the domain model. The domain elements are the entities and value objects. The domain services use the elements as indicated by the arrows. Finally, you compose the domain services to implement a few higher-level domain behaviors.

interact with the other modules and data structures of our model. From our implementation, you’ll consider `AccountService` as the candidate module to elaborate on. You’re familiar with most of the implementation of this module by now.² Still, we’ll introduce new types in this implementation, make it more domain-friendly, and integrate many of the concepts introduced separately in earlier chapters. This is a detailed exercise, and you’ll explore this module incrementally through the next few sections.

THE PUBLISHED ALGEBRA

You’ve seen what I mean by algebraic API design.³ All modules that you define will have algebra that serves as the binding contract with the users of the module. Listing 5.1 details this algebra for `AccountService`. Let’s look at what constitutes a module from this contract point of view:

- *Name*—`AccountService` is a name that resonates with the domain vocabulary. You’ll proceed with this as the name of the module.
- *Operation names*—You use `open`, `close`, `debit`, `credit`, and so forth as the operation names. These again come straight from the ubiquitous language of the domain.
- *Trait as the container*—You use traits as the container of the module definition. Traits in Scala allow you to define operations (and optional implementations) and mixin composition with other modules.

² Remember, we’ve been discussing this domain service since chapter 3.

³ We discussed algebraic API design in chapter 3.

- *Type aliases*—Anything you publish in a module should have domain-friendly names. Types are no exception. You define domain-friendly type aliases to hide complex implementations underneath.
- *Parameterized*—Whenever possible, define modules with parameterized types. You can then provide your own concrete types when you implement the module.
- *Compositionality*—All operations of the module return the type `AccountOperation`, which is an abstraction over evaluation and not a value.⁴ This helps in making your functions more compositional. Look at the definition of the `transfer` method, which uses composition of types to define a new operation *only* from the algebra of other operations.
- *Implementation independence*—The entire module definition has no implementation details in it. You already know from chapter 3 why you need to separate the algebra from the implementation when defining abstractions.
- *Making collaborations explicit*—If you need to collaborate with other patterns from within your module, make it explicit so that the user is aware of the overall scope of your module. In `AccountService`, you make it explicit that you need the services of `AccountRepository` (the Repository pattern covered in chapter 3) in order to deliver account management functions. Also the type definitions make it explicit that you use dependency injection through the `Reader` monad in order to make the services of `AccountRepository` available within your module.⁵

Listing 5.1 Algebra of the module `AccountService`

Module definition—domain-friendly name and parameterized on types

NonEmptyList ensures statically that you can't have an empty list when you're on the left data constructor of the disjunction. You keep a List here instead of a single string because the underlying implementation of the service may provide a list of errors in case of failures. As discussed in chapter 4, this is possible with the applicative model of effect handling. The code repository for this chapter has examples that do this. The sidebar that follows this section also describes the choice of a right-biased Either from Scalaz as the return type.

```
trait AccountService[Account, Amount, Balance] {
    type Valid[A] = NonEmptyList[String] \/\ A
    type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

    def open(no: String, name: String, rate: Option[BigDecimal], ←
            openingDate: Option[Date], accountType: AccountType): ←
        AccountOperation[Account]

    def close(no: String, closeDate: Option[Date]): AccountOperation[Account]
    def debit(no: String, amount: Amount): AccountOperation[Account]
```

Operation definition with domain-friendly name

⁴ We discussed the difference between an abstraction over evaluation and a value in chapter 3.

⁵ We use `Kleisli` here instead of the `Reader` abstraction discussed in chapter 3. They're equivalent; it's a good exercise to try to prove that the `Reader` monad can be implemented in terms of the `Kleisli`. `Kleisli` is discussed in chapter 4.

```

def credit(no: String, amount: Amount): AccountOperation[Account]
def balance(no: String): AccountOperation[Balance]

def transfer(from: String, to: String, amount: Amount):
    AccountOperation[(Account, Account)] = for {
        a <- debit(from, amount)
        b <- credit(to, amount)
    } yield ((a, b))
}

Composability—transfer is defined as a composition of other operations. This is possible because you have return types as computations for the methods.

```

Figure 5.3 annotates the various parts of a module definition for AccountService. It elides the operations, but highlights the issues that you need to remember when you define a module on your own.

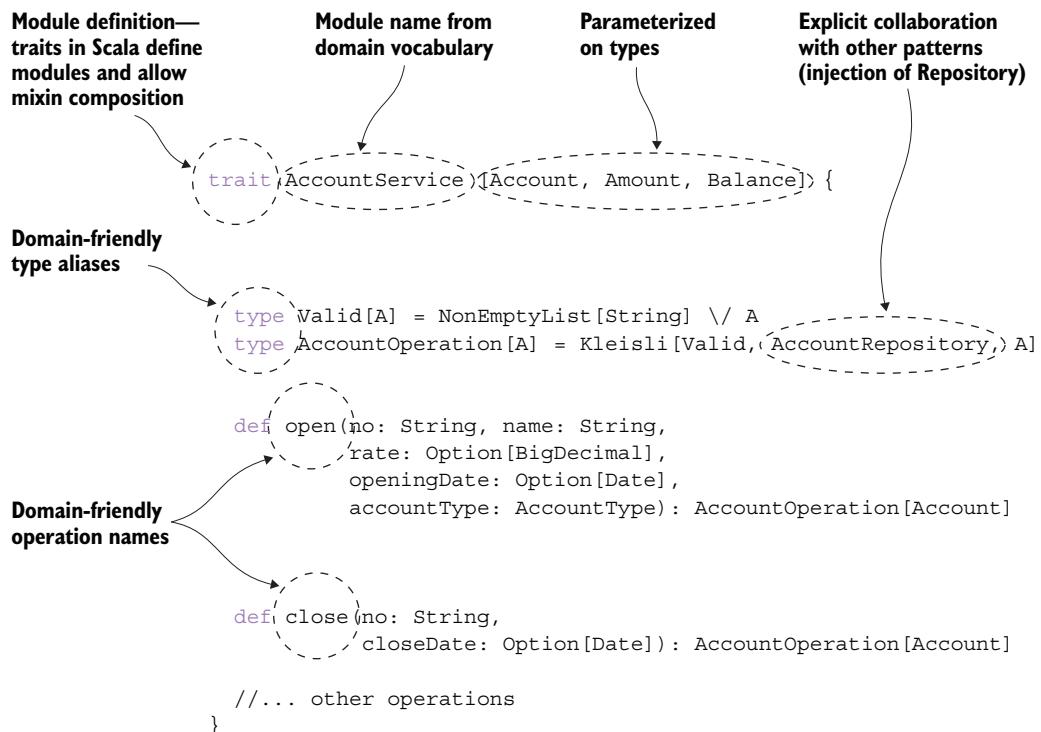


Figure 5.3 Annotation of the important parts of a module definition, which are discussed in the accompanying text. This figure shows what you need to check when you define a module.

THE MODULE IMPLEMENTATION

An implementation of a module is the interpreter of its algebra. You know by this time that you can have multiple implementations for the algebra. Various parts of your application can use any of these implementations. And as a designer, it's always a good software-engineering practice to *commit to specific implementations as late as possible*.

Figure 5.4 illustrates the dependencies that a module algebra and implementation induce in your model.

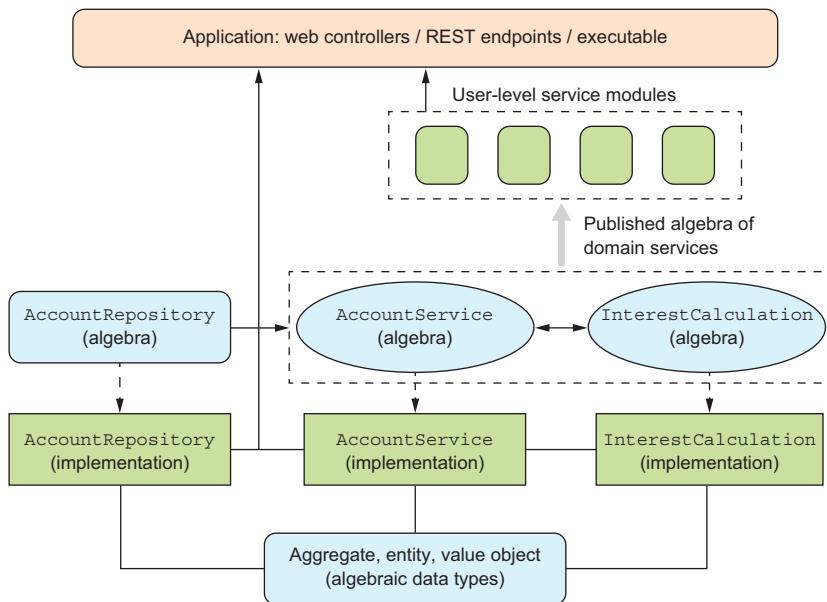


Figure 5.4 The dependency structure that a module induces within your model. An arrow from A to B indicates that B uses A. The only part of your model that uses the module implementation is the end-user application. The rest of the model works solely based on the module's algebra.

Here, the *Application* box is the end-user artifact. Note its dependencies. *Application* is the only context that depends on the service *implementations*.⁶ Every other abstraction in the model depends on the algebra. None of them leaks implementation to other modules. And this is the most important aspect of modular design that you need to know. If there's one point that you can take away from this section, it's that you need to be careful about leaking implementation details of your modules. If your implementation leaks out, modules become coupled with each other. This prevents independent evolution of module implementations.

In the dependency structure of figure 5.4, *AccountService* depends on *AccountRepository*, but this dependency is only at the level of the algebra. In a production system, you may have an implementation of *AccountRepository* based on an enterprise database. But while testing, you can swap it out and inject some other lightweight, in-memory implementation. You can do this because the module *AccountService* is

⁶ More on this in section 5.2.2.

totally decoupled from the implementation of the AccountRepository. The complete implementations of all the modules are available as part of the online code repository for this book. The following listing shows a sample implementation of a few methods of the interpreter for AccountService.

Listing 5.2 Interpreter for AccountService

```

package domain
package service
package interpreter

import java.util.{ Date, Calendar }

import scalaz._  
import Scalaz._  
import \/_.  
import Kleisli._

import model.{ Account, Balance }  
import model.common._  
import repository.AccountRepository

```

Implementation is in subpackage interpreter

```

class AccountServiceInterpreter extends
  AccountService[Account, Amount, Balance] {

  def open(no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType) = kleisli { (repo: AccountRepository) =>

```

Kleisli for injecting repository

```

    repo.query(no) match {
      case \/-(Some(a)) =>
        NonEmptyList(s"Already existing account with no $no").left[Account]
      case \/-(None)     => accountType match {
        case Checking =>
          Account.checkingAccount(no, name, openingDate, None,
            Balance()).flatMap(repo.store)
        case Savings   => rate map { r =>
          Account.savingsAccount(no, name, r, openingDate, None,
            Balance()).flatMap(repo.store)
        } getOrElse {
          NonEmptyList(s"Rate needs to be given for savings
account").left[Account]
        }
      case a @ -\/(_) => a
    }
}

```

Pattern match on right of Disjunction—in Scalaz right is \/-

Imports model elements and repository algebra

Smart constructors for checking and savings account creation

Returns the left of the disjunction as a NonEmptyList

```

def close(no: String, closeDate: Option[Date]) =
  kleisli { (repo: AccountRepository) =>
    repo.query(no) match {
      case \/-(None) => NonEmptyList(s"Account $no does not
        exist").left[Account]
      case \/-(Some(a)) =>
        val cd = closeDate.getOrElse(today)
        Account.close(a, cd).flatMap(repo.store)
      case a @ -\/(_) => a
    }
  }

//... other operations
}

object AccountService extends AccountServiceInterpreter

```

And here are a few notes on the implementation that you may find different from the earlier version developed in chapters 3 and 4:⁷

- *Kleisli*—You use Kleisli for injection of AccountRepository instead of the Reader monad. Semantically they’re the same; you can implement Reader in terms of Kleisli (and that’s exactly what Scalaz does).⁸ The benefit of using Kleisli is that you have access to helpful combinators (that you also saw in chapter 4).
- *Right-biased Either*—The return type of your service methods is a right-biased⁹ Either type from Scalaz. It’s named \/ and offers convenient infix syntax as well. \/ is a monad and offers convenient combinators for transformations into a host of Scala standard classes such as Either and Option. Until now, you’ve seen abstractions being used for validation (for example, use cases where failure is one of the valid options—Try, Either, and scalaz.\/). Each has its own set of pros and cons, summarized in the following sidebar.
- *Smart constructors*—Note the use of smart constructors for creating checking and savings accounts. As you saw in chapter 4, this prevents implementation of the subtypes of Account from leaking into service implementations.
- *Dependency*—The service implementation depends on the model elements (Account, Balance, and so forth) and the *algebra* of AccountRepository (not the implementation). Look at the complete implementation of the model elements in the online code repository.

⁷ And these differences are improvements that you’re making incrementally.

⁸ Scalaz: <https://github.com/scalaz/scalaz>

⁹ *Right-biased* means that all the combinators such as map will work on the right projection of the type. This is helpful when you use \/ for validation; the right side is assumed to hold the validated value on which the higher-order functions can be mapped over.

scala.util.Try and variants—which one to use?

For handling validation-like uses when using Scala, you have many out-of-the-box options at your disposal. A few come from the Scala standard library, and a few others come from Scalaz. Let's summarize the uses (pros and cons) for each of them, so that as a designer you can make an informed decision on which one to choose for your model.

- `scala.util.Try`. Try is tailor-made to be used with APIs that can throw exceptions. The `Failure` data constructor of `Try` takes a `Throwable` as the argument. This may seem to go against the core principle of FP, which doesn't encourage exceptions to be leaked out of your function. But you may find `Try` to be a useful construct when dealing with external libraries (especially Java libraries) that throw exceptions. Another important factor that drives the adoption of `Try` is that it's part of the standard library. In various parts of this book, I've mentioned that `Try` is a monad because it has a `flatMap`. But strictly speaking, `Try` violates one of the laws of functor composition, as has been reported in SI-6284 (<https://issues.scala-lang.org/browse/SI-6284>). This may seem to be of theoretical interest and is unlikely to impact its adoption for most use cases. Still, it's useful to know that `Try` is an abstraction that handles exceptions as effects even in the world of FP and at the expense of some basic laws of functor composition.
- `scala.util.Either`. You can use `Either[Throwable, A]` instead of `Try[A]`, and it will have the same impact. But with `Try` you get some nice combinators out of the box, which you may have to hand-code with `Either`. Also `Either` isn't a monad, though you can use it as a monad through `LeftProjection` or `RightProjection`. It's a bit cumbersome, though.
- `scalaz._/.` This is a right-biased variant of `scala.util.Either`. This is part of Scalaz and is a monad. It has useful combinators and offers seamless interoperability with `scala.util.Either`. For implementing validation logic that has monadic effects, this is an extremely useful abstraction (see chapter 4 for the differences between monadic and applicative effects).
- `scalaz.Validation`. As the name suggests, this is the most generic and powerful technique for handling validations. `Validation[E, A]` is an applicative (discussed in chapter 4) and offers built-in support for accumulating errors through a Semigroup implementation of `E`. When you want to accumulate all errors and report them at once from your API, use this abstraction. Use `scalaz._/_` if you need monadic sequencing.

5.2.2 Composition of modules

As you've seen before, and as discussed earlier, modules in Scala compose via mixin-based composition. This is one of the reasons we say that Scala offers first-class support for modules. Here's an example from our domain of personal banking related to the small subset discussed in this section:

```
trait InterestCalculation[Account, Amount] {  
    def computeInterest: Kleisli[Valid, Account, Amount]  
}
```

```
trait TaxCalculation[Amount] {
    def computeTax: Kleisli[Valid, Amount, Amount]
}
```

InterestCalculation and TaxCalculation are separate modules that offer algorithms for computing interest and tax, respectively, on the client balance in their accounts. But often you'll want to compose them together and provide localized algorithms for both of them (possibly honoring country-specific computation rules). You can define a larger module composing InterestCalculation and TaxCalculation:

```
trait InterestPostingService[Account, Amount]
  extends InterestCalculation[Account, Amount]
  with TaxCalculation[Amount]
```

And then you can provide a combined interpreter for the composed algebra:

```
class InterestPostingServiceInterpreter extends
  InterestPostingService[Account, Amount] {
    def computeInterest = ... implementation
    def computeTax = ... implementation
}

object InterestPostingService extends InterestPostingServiceInterpreter
```

You can find the entire implementation in the online code repository of the book.

5.2.3 Physical organization of modules

Now that you've seen the virtues of organizing your model into modules and the benefits that you get from decoupling the implementation of a module from the algebra, let's focus on how to organize the package structure so that you get correct modularization. Proper packaging leads to proper visibility of abstractions, which is also an important aspect of writing resilient code.

The strategy presented in this section is just one example of how to organize your model into modules. Depending on the complexity of your model, feel free to make changes. The core idea is to appreciate the advantages that a well-modularized code base brings to your model. Before discussing the structure, let's look at figure 5.5, which illustrates the entire structure for one single module. It shows how to organize the module's algebra, its implementation, the collaborating modules, and finally the end-user application.

You have the base package as `domain`, under which you have the following subpackages, each containing specific artifacts in the form of separate modules:

- *Model elements*—The package is named `model` and contains the algebraic data types modeling entities, aggregates, and value objects. It also contains the companion objects that will have the smart constructors, lenses, and other supporting abstractions for the core model elements.

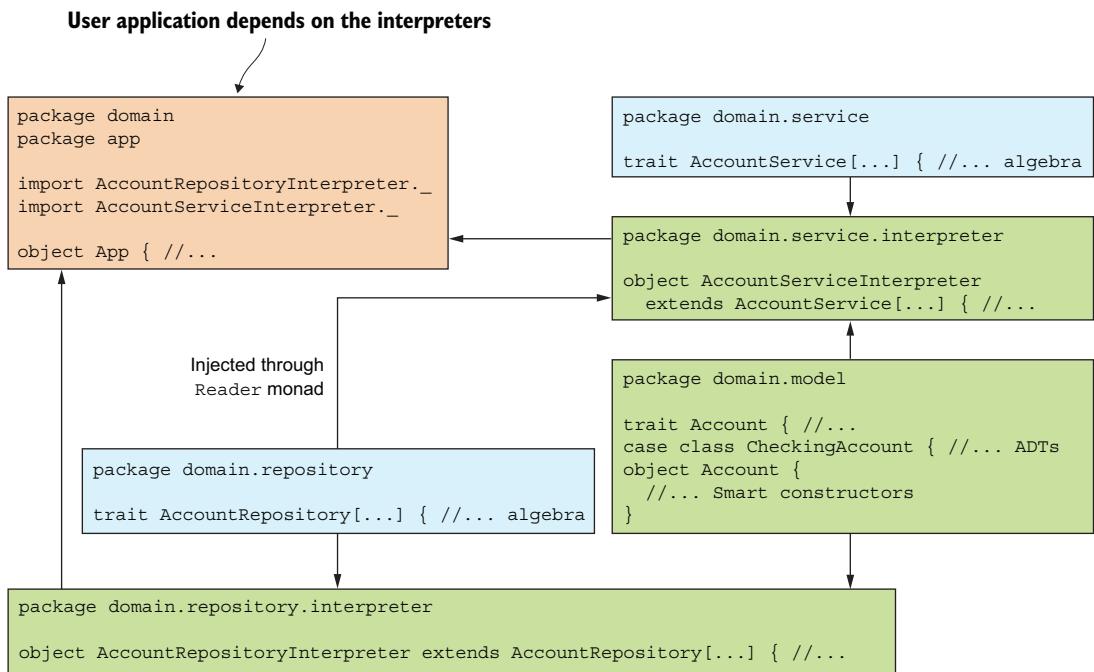


Figure 5.5 Organizing a module along with the collaborators. Every arrow indicates a dependency between modules. The figure shows the domain elements in package model, the service module algebra in the package service, and the interpreters in the package interpreter. The interpreters are used only in the end-user application, denoted by App and residing in the package app.

- *Repositories*—Repositories stay within separate modules, and you name this package repository. Because a repository makes a module, you’ll have implementations as well. So the algebra stays within repository, whereas the interpreters can be placed within the subpackage interpreter or implementation.
- *Domain services*—The organization is exactly that of repositories. You have a package, service, for the algebra, and interpreter for all the implementations.
- *Application*—This is the package for the end-user application named app. In the current context, it’s just a placeholder. In real life, it’ll have the package structure that your end-user application will mandate. If you’re using a framework such as Play, you’ll have a package structure that conforms to the structure of a Play application.

This is just an example of what a package structure may look like when you have all the artifacts of domain-driven design. Based on the complexity of your model, you can always make changes. I’ve kept all service modules at one level. If you have lots of services within the same bounded context, you may need to introduce subpackages based on functionalities. Another approach is to split into subpackages at the top level if you have different functional modules within the same bounded context. But

the important thing is to control the visibility and keep a strong check so that the implementation isn't inadvertently leaked out. This will make your model extremely difficult to evolve.

5.2.4 Modularity encourages compositionality

The preceding section introduced a package named `app`, in which you'll have the client application. And this is the only place where you'll bring in the respective implementations of your modules. This is per the guideline that you should *commit to the concrete implementations only at the end*. The following listing shows a sample client composition from our continuing example.

Listing 5.3 Using the modules in the client application

```
import AccountService._           ① Imports all
import InterestPostingService._  implementations

def postTransactions(a: Account, creditAmount: Amount, debitAmount: Amount)
    Kleisli[Valid, AccountRepository, Amount] = for {
        _ <- credit(a.no, creditAmount)
        d <- debit(a.no, debitAmount)
    } yield d

def composite(no: String, name: String, creditAmount: Amount, debitAmount: Amount): Kleisli[Valid, AccountRepository, Amount] = (for {
    a <- open(no, name, BigDecimal(0.4).some, None, Savings)
    t <- postTransactions(a, creditAmount, debitAmount)
} yield t) andThen computeInterest andThen computeTax           ② Composes methods
                                                               from a module

val x = composite("a-123", "John k", 10000, 2000)           ③ Inter-module
                                                               composition
                                                               implementation for Account-
                                                               Repository (see online code
                                                               repository for an implementation)
```

Here are some salient points about this implementation:

- *Committing to module implementations*—This is the place where you need to use the concrete implementations of the modules. The first two imports ① in the code listing do exactly this. Also when you invoke the `composite` command ②, you feed it the implementation for the `AccountRepository`.
- *Composing methods from a module*—In the `postTransactions` function, you compose methods from `AccountService`. You can do this because each method returns a monad. This is the value that you get by returning *effects* from the methods rather than concrete values.
- *Composing across modules*—When building larger services or abstractions, it's common to use multiple modules as part of the implementation. One trick to help you build composable abstractions is to keep an eye on the types that the module methods return. In the `composite` function, you compose methods from `AccountService` as well as `InterestPostingService`. The `for`-comprehension in `composite` returns a `Kleisli`, which can be naturally chained into methods

from the module `InterestPostingService`, which also returns Kleislis. Types align, and you get compositional bliss. The more you encapsulate within your types, the less noise will surface in your implementation. The function `composite` is a good example, illustrating this quality of abstraction composition.

5.2.5 Modularity in domain models—the major takeaways

All discussions in section 5.2 point to the fact that well-structured software needs to be composed of a collection of modules. Each module needs to be cohesive but minimally coupled to other modules and the environment. You learned a lot of things regarding the contracts that modules need to publish, and how they should be implemented and organized as part of your code base. Here are the main takeaways regarding well-modularized domain models:

- *Published contracts*—A module needs to have clearly defined contracts with the client. These are the published interfaces of the module (also known as the algebra), which can't be changed without disrupting existing clients.
- *Private implementation*—A module's implementation should be as private as possible and shouldn't be leaked into client code.
- *Module organization*—You organize module contracts as Scala traits and compose depending on domain semantics. Commit to the implementations only at the client application level, sometimes called the *end of the world*. Keep modules in separate packages and segregate the module algebra from the implementation at the package level.
- *Compositionality*—Clients should be able to compose modules together and evolve larger modules from smaller ones.

5.3 Type class pattern—modularizing polymorphic behaviors

When designing domain models, you'll frequently need to implement specific behaviors across many objects. These objects may not be related in any way, but may need to share this common behavior. A common example of this is serializability. Many domain objects such as `Account`, `Customer`, or `Bank` may have to be serializable when you need to pass them across multiple contexts. You'll see one such use case in section 5.4.2 when we discuss communication between multiple bounded contexts.

One way to implement this using the object-oriented paradigm is by using subtype polymorphism. You have a trait for the base behavior, and then every class can provide an implementation for that specific behavior. In Java, you have `java.io.Serializable`, which all of us have at some point used to provide class-specific serialization behavior. Here's one sample implementation approach in Scala:

```
trait Serializable[T]
trait Account extends Serializable[Account] { ... }
case class Customer(...) extends Serializable[Customer] { ... }
```

This approach has several disadvantages. First, it couples the core domain abstraction with all such noncore behaviors that it may have to implement. Second, you need to specify this behavior at the site of the class definition. You must have access to the source code of `Account` in order to make it serializable. Some alternative patterns can overcome this drawback, but they're extremely verbose and cumbersome to implement and maintain.

Type classes offer an alternative approach to this problem. The basic idea is to make classes that have no relationship between them behave *polymorphically* with respect to specific behaviors. This is sometimes known as *ad hoc* polymorphism and is implemented using the power of parametric polymorphism.

Let's consider a small example from our domain of personal banking. When you need to generate reports, you display domain elements in a specific format. For some financial entities such as a security or a currency, there's ISIN code, which has a specific format. Then customer accounts need to be displayed in a format that may differ across financial institutions. In summary, you want to design a protocol for showing elements (behaviors and objects) from your domain models; let's name this protocol `Show`. `Show` defines a method `shows` that needs to be implemented for every domain abstraction for which you need a custom display protocol:

```
trait Show[T] {
    def shows(t: T): Try[String]
}
```

But how do you implement this behavior across multiple unrelated abstractions without changing the definitions? This is what the *Type Class* pattern offers. The Type Class pattern was first implemented in Haskell, and Scala also offers a way to encode this pattern. Let's explore how to do this encoding for our `Account` class. The same technique can be extended to other classes as well:

```
case class Account(no: String, name: String, dateOfOpening: Date = today,
    dateOfClosing: Option[Date] = None,
    balance: Balance = Balance(0))
case class Customer(no: String, name: String, address: Address,
    email: String)
trait ShowProtocol {
    implicit val showAccount: Show[Account]
    implicit val showCustomer: Show[Customer]
    ...
}
```

You have the class `Account` and then you define a module that defines the algebra for the `Show` protocol for the domain elements. The next step is to provide implementation for each of these protocol elements:

```
trait DomainShowProtocol extends ShowProtocol {
    implicit val showAccount: Show[Account] = new Show[Account] {
        def shows(a: Account) = Success(a.toString)
    }
}
```

```

implicit val showCustomer: Show[Customer] = new Show[Customer] {
    def shows(c: Customer) = Success(c.toString)
}
//...
}
object DomainShowProtocol extends DomainShowProtocol

```

Now you have a module, `DomainShowProtocol`, that implements the algebra you defined as part of the protocol in `ShowProtocol`. You'll soon see why you've made the implementations implicit.

You've defined the protocol implementations in a separate module from the domain elements. The elements are completely decoupled from the protocols, and this gets rid of the problems that we talked about with the OO implementation. Let's now look at how to use the Type Class pattern:

```

object Reporting {
    def report[T: Show](as: Seq[T]) = as.map(implicitly[Show[T]].shows(_))
}

```

Say you have a concrete reporting module with a `report` function. `report` displays the collection of elements passed to it according to a specific protocol. You can pass specific implementations of the `Show` protocol and expect `report` to display elements accordingly. In this example, you've used Scala's context-bound syntax to make the `Show` protocol an implicit argument. When you invoke the `report` function, the compiler will automatically pass an appropriate implicit value for the protocol if available in scope.¹⁰ The following example imports an implementation module for the `Show` protocol, which gets automatically passed in when you call `report`. You use the protocol encoded for the `Show` type class to make the API more succinct and type safe. The compiler will complain if it's not able to find any matching implicit value in scope when you invoke `report`.

```

import DomainShowProtocol._

val as = Seq(
    Account1("a-1", "name-1"),
    Account1("a-2", "name-2"),
    Account1("a-3", "name-3"),
    Account1("a-4", "name-4")
)
Reporting.report(as)

```

1 Imports default implementation of all domain protocols for Show

2 Uses default protocol implementation

In summary, what are the advantages of using type classes in domain modeling? Here are the main ones:

- *Modularity*—Type classes group related behaviors into modules that can be exported for implementation across domain elements. They make your code

¹⁰ *Scope* indicates the scope within which the protocol is visible. This is implemented in terms of implicit parameters in Scala. See Implicit Parameters and Views at www.scala-lang.org/files/archive/spec/2.11/07-implicit-parameters-and-views.html for the detailed scope resolution for implicits in Scala.

modular by grouping related behaviors instead of grouping related objects. This philosophy goes well with the principles of functional programming.

- *Ad hoc polymorphism*—Using type classes, you can implement polymorphism for selected behaviors across unrelated abstractions. In the preceding example, you implemented a protocol that creates a custom display for domain elements such as `Account` and `Customer`, which aren't related in any way. And the best part is you can implement such behaviors on existing abstractions, which you can't do with subtype polymorphism.
- *Protocol selection*—In the preceding example, you imported the default implementation of all protocols ① to be implicitly used in the invocation of `report` ②. Following Scala's rules of implicit resolution, you could also provide an alternative implementation of any protocol when you invoke the `report` method. This gives you the power to have layers of encoding of the type class for protocol implementation and plug in the one that suits the context best.

5.4 Aggregate modules at bounded context

Chapter 1 covered bounded contexts in terms of domain-driven design and its various patterns and artifacts. Let's dive a little deeper and discuss the role that bounded contexts play in the modularity of your domain model. The bounded context is possibly the most important concept in modularizing complex domain models. Any nontrivial domain model is not really one model: It consists of multiple models. Even the domain of personal banking consists of multiple smaller models, such as account management, reporting services, and back-office management. These are different functionalities having completely different sets of entities, behaviors, and vocabulary and may have to be implemented using completely different domains and data models. Even artifacts that are named the same way in these models may mean completely different things with different lifecycles.

Consider the abstraction `Account`. When you think of `Account` in the core banking model, you know it's an entity (you've seen this many times in earlier chapters). An account's lifecycle needs to be managed, with the account number providing the identity of the account. In a reporting context, you don't manage accounts as entities; `Account` in a reporting module is a value object whose values need to be printed and managed through a denormalized data model for faster queries and report generation. Every application needs to have an authentication module, and `Account` in the context of authentication has a completely different meaning. It's a user account that needs to be authenticated for system usage and is completely different from a customer account that the bank maintains.

Each of the smaller models within the larger model forms a *bounded context*, as you'll see here. Each bounded context

- Has a single coherent data and domain model
- May have elements with similar names as other bounded contexts but with completely different semantics
- Has a ubiquitous language that's valid only within that specific context

- Needs to have minimal coupling with other bounded contexts, which has to be explicitly defined

We've only recently discussed modules in our model. Why not think of functionalities as separate modules? A model needs to be strongly consistent within a single bounded context. You can't have two elements with the same name and different semantics within a single bounded context. That's an ambiguity in the data model and clearly screams for a separate bounded context.

NOTE For more details on how the bounded context relates to the practice of domain-driven design, look at *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley Professional, 2003) and *Implementing Domain-Driven Design* by Vaughn Vernon (Addison-Wesley Professional, 2013).

5.4.1 Modules and bounded context

How are modules and bounded contexts related? A bounded context represents a higher level of granularity and typically contains multiple modules within it. Figure 5.6

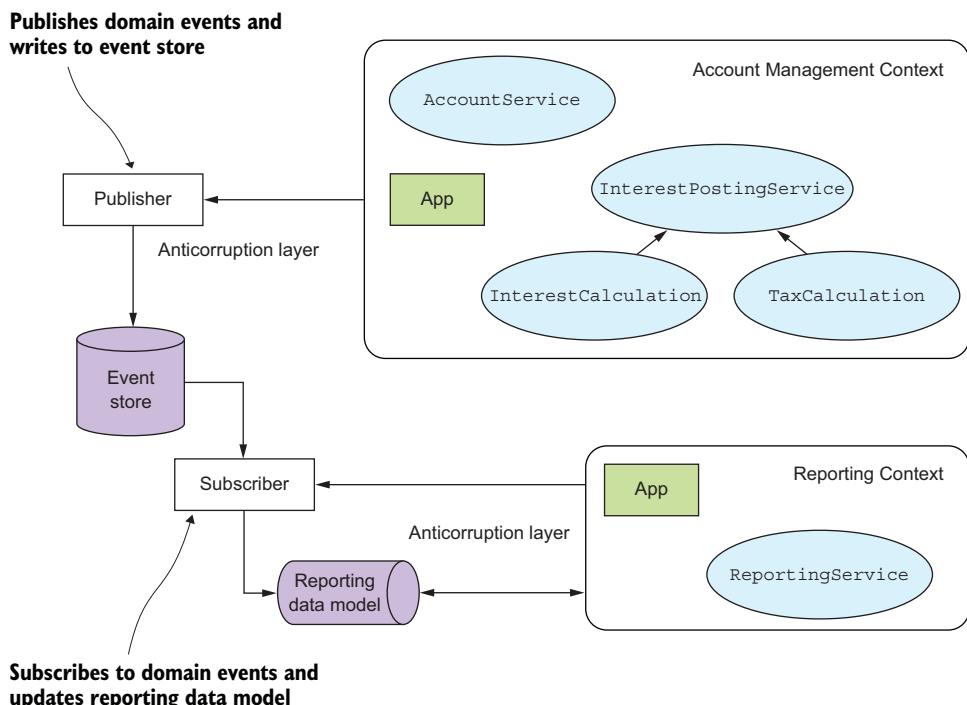


Figure 5.6 A bounded context contains multiple modules. Account Management and Reporting are two bounded contexts that have multiple modules. The rectangular Publisher and Subscriber boxes denote the context map and set up the communication protocol between the two bounded contexts. There's no direct communication between the two bounded contexts. All communication is through explicitly published protocols specified in the context map.

clearly explains the relationship in the context of our running examples. There are a few annotations regarding communication between bounded contexts, which we'll discuss soon.

5.4.2 **Communication between bounded contexts**

In *Domain-Driven Design*, author Eric Evans notes a few ways of setting up a communication pathway between two bounded contexts. One is through an *anticorruption layer*, which sets up an explicit layer of communication between the two bounded contexts. This prevents one bounded context from messing directly with the implementation of another and provides isolation between the two components. Besides the anticorruption layer, Evans presents a few more techniques in his book, which you can go through for more details. But the basic idea is to ensure that there's no uncontrolled communication across bounded contexts.

Semantics of abstractions can break when you communicate between two bounded contexts, as you saw with the example of Account in the context of account management and reporting services. In both contexts, the name is the same, as it's derived from the domain vocabulary. But the attributes change, the lifecycles change, and identity management changes. Clearly the Account abstraction in account management isn't the same as the one in a reporting context. But they're related, and you need to do this mapping when you want reporting services to print the details of accounts that were created in the account management context.

Messaging is one of the most powerful techniques for this mapping without coupling the two contexts. Messaging provides a natural mapping place for types, where you deconstruct your abstractions and map one to the other. Another advantage of messaging is that making it asynchronous gives you decoupling in space and time. The two contexts can be temporally and spatially decoupled, and yet asynchronous messaging can serve as the perfect glue for communicating across them. The only requirement is that you need to have a protocol that both contexts understand and respond to the messages they receive.¹¹

In this example (see figure 5.6), you've used the publish-subscribe model of interaction as the messaging protocol for designing the anticorruption layer. The account management context executes commands for which events are published and stored in the event log. The reporting context is a subscriber to these events and updates its repository on receipt of any such event. This can trigger an online generation of reports based on the new data received. Here you have two bounded contexts, each operating within the constraints of its data and domain model, and carefully protecting the sanctity of its invariants through an explicitly published communication layer. Chapter 6 describes a sample implementation of this use case.

¹¹ Thanks to Martin Thompson for articulating this idea nicely on Twitter.

5.5 Another pattern for modularization—free monads

The main purpose of modularization is to ensure that you can change one part of your model without any impact (or minimal impact) on other unrelated parts. You ensure this by building proper abstractions. A module is such an abstraction, which also acts as the container for other abstractions. You saw this with our example of `AccountService` earlier in this chapter. For a quick recap, a module

- Is a collection of types and functions with explicitly specified algebra
- Models domain behaviors that are semantically related
- Can have multiple interpreters or implementations that need to be decoupled from the algebra

You saw examples of how to modularize your domain models in section 5.2. You learned about `AccountService`, a module handling account management functionalities. You saw the algebra that it publishes and took a look at a sample implementation.

`AccountService` uses another module, `AccountRepository`, which offers functions that the `Account` aggregate uses to interact with the repository. This section focuses on this repository and discusses an advanced pattern to separate the contract of the module from its implementation. You'll first take a look at the standard interface/implementation segregation pattern that we've been using so far. And then you'll try to take it to a different level by using advanced techniques of functional programming.

NOTE This is an advanced topic, so feel free to skip this section in your first reading of this chapter.

5.5.1 The account repository

Let's start with the contract of this module, one that is published to the clients and used by `AccountService`.

Listing 5.4 The `AccountRepository` module definition

```
Uses the domain elements
Account and Balance
import model.{ Account, Balance }

trait AccountRepository {
    def query(no: String): \/[NonEmptyList[String], Option[Account]]
    def store(a: Account): \/[NonEmptyList[String], Account]
    def balance(no: String): \/[NonEmptyList[String], Balance] =
        query(no) match {
            case \/- (Some(a)) => a.balance.right
            case \/- (None) => NonEmptyList(s"No account exists with no
$no").left[Balance]
            case a @ -\/(_) => a
        }
    def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]]
}
```

NonEmptyList is a list abstraction with a constraint that it can't be empty. Always consider using such abstractions that encode some constraints implicitly.

The module definition—each method returns the right-biased Either of Scalaz

A module definition can have multiple implementations, so let's start with the most obvious one that implements the preceding trait and provides the semantics to each of the method definitions by using an in-memory mutable Map.

Listing 5.5 A sample in-memory implementation of AccountRepository

```
import scala.collection.mutable.{ Map => MMap }
import model.{ Account, Balance }

trait AccountRepositoryInMemory extends AccountRepository {
    lazy val repo = MMap.empty[String, Account]

    def query(no: String): \/[NonEmptyList[String], Option[Account]] =
        repo.get(no).right
    def store(a: Account): \/[NonEmptyList[String], Account] = {
        val r = repo += ((a.no, a))
        a.right
    }
    def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]] =
        repo.values.filter(_.dateOfOpen == openedOn).toSeq.right
}
object AccountRepository extends AccountRepositoryInMemory
```

Now that you have the interface of the module and an implementation, you can run the following snippet and get some results:

```
import AccountRepository._
import scalaz.syntax.std.option._

val account = checkingAccount("a-123", "John K.", today.some,
    None, Balance(0)).toOption.get
val dsl = for {
    b <- updateBalance(account, 10000)           ←
    c <- store(b)
    d <- balance(c.no)
} yield d
```

Smart constructor for creating a checking account

updateBalance is a function in Account module that updates the balance of an account and returns NonEmptyList[String] \ Account.

You have an implementation of the module `AccountRepository` with the semantics of each method defined. Each method of the module returns a disjunction (`scalaz.\/()`), which is a monad. Hence it's no surprise that you can create your little DSL with for-comprehensions to create accounts, store them, query and fetch balances, and so forth using `AccountRepository`.

Running the preceding code results in `\/-(Balance(10000))`. Because `\/` is a monad, this sequence uses `flatMap` and `map` to thread through the computation and finally collapses the monadic context to yield the result.

5.5.2 Making it free

The implementation in section 5.2.1 is fine and solves most problems of decoupling the interface from the module implementation in your model. This section presents a new way of decomposing the functionality of a module: You'll model domain behaviors

as *pure data* and then supply *interpreters* that work on that data in specific contexts. This is much like the way compilers work; you transform your program into an abstract syntax tree (AST), which is pure data, and then define separate interpreters that manipulate the tree and perform various transformations such as optimization, compilation to byte code, and pretty printing.

When you design a module for a specific set of domain behaviors, think of each of the behaviors modeled as pure data without any concern for how it will be executed operationally. Once you've identified all the behaviors of the module, group them together as a closed algebraic data type. With an ADT, you can now compose the elements to form complex recursive values that model more-complex domain behaviors. This is all about creation of computation structures, and you've not yet seen a wee bit of implementation regarding how the data types will be executed. Execution comes in the form of interpretation and is a separate stage altogether in this pattern of modularization. After you have the data types, you can define multiple interpreters, each with its own semantics of execution. You may execute them directly, you may perform some optimizations by applying domain rules, or you may pretty print all the data for audit trails.

We call this the *Free Monad* pattern. You'll see examples of implementation in later sections. But just as a summary of what you've learned so far, here are the salient features of a free-monad-based computation structure:

- Behaviors represented as pure data forming a closed algebraic data type.
- Strict separation between creation of the computation and its execution.
- Execution of computation comes in the form of interpreters, and you can have multiple interpreters for the same structure.

Now that you have an understanding of how free monads help modularize your model, let's take the next step and find out how to apply this pattern in practice. Specifically, you'll learn about the steps to arrive at the separation between pure data and the interpretation for your module, assuming you have the machinery of free monads at your disposal. I'll leave some steps intentionally fuzzy but we'll come back to these later.

- *Building blocks*—You start with a bunch of behaviors that model individual operations forming the algebra of your module. These are the building blocks that you can compose to form larger behaviors or even a little DSL for your module. In the context of `AccountRepository`, you can think of the individual operations as forming the building blocks of your DSL. This is the same idea that we used in earlier implementations of `AccountRepository`. The difference is in how you model these behaviors; they're no longer modeled as individual functions in the module. You design them as pure data using algebraic data types. You'll see examples shortly.
- *The magic band*—One of the ways you can compose the building blocks in sequence is to have them within a monadic context. But you don't have a monad

now; you have only some ADTs as pure data. Here's where the magic band comes in: You do something magical and get a *monad for free*. Don't worry about the details for now. You'll see how the magic works in the next section. For now, assume that somehow you get a monad within which you can lift each of your building blocks.

- *The module*—You have a free monad and now you need to lift each of your building blocks into the context of the monad. These lifted operations form the public API of your module. Because each operation now returns a monad, you can compose them to form your DSL using for-comprehensions.
- *What does the composed DSL return?* After you have your DSL using for-comprehensions, you expect to execute it by threading through the `flatMap` (bind) of the monad. What the sequencing of a monad does is defined by what the `flatMap` of the monad implements. And it so happens that the `flatMap` of your free monad does nothing; it just accumulates the building blocks as pure data and hands over the resulting aggregate (which is the AST). It's for this reason we say that the free monad has no semantics or denotation.¹² So now you have the first step complete; your free monad has handed over the pure data with no context whatsoever.
- *Denotation*—Now that you have the dumb AST, you need to supply an interpreter to make it execute according to your requirements. The free monad has done the accumulation part (which the `flatMap` does), but didn't execute the collapsing of the monadic context. This is how the `flatMap` handed over the AST itself. The interpreter traverses the AST and defines the semantics of the DSL. The interpreter has the context of application. If you want your AST to be executed and the result returned as a disjunction of error and values, you can implement the interpreter that way. If you want to execute the AST asynchronously, you can return a `Future` from your interpreter. The interpreter applies the context on top of the pure data that the free monad gives you.

The preceding steps introduce a layer of indirection between the definition of the DSL structure and the interpretation. In the earlier example in section 5.2.1, the DSL for composing sequences of operations from `AccountRepository` already had its execution sequence defined by the monadic context of `scalaz.\/`. The interpreter provided the implementation of the abstract methods. In this case, the free monad leaves the execution sequence undefined, and it's up to the interpreter to fix that, traversing through the AST.

5.5.3 **Account repository—monads for free**

You could start by looking into the details of implementation of a free monad. But as a user, you'll hardly have to worry about most parts of it. So let's try to apply the first

¹² Remember it was mentioned that all semantics would be in the interpreter.

four steps listed in section 5.5.2 to our use case of implementing AccountRepository as a free monad. You'll keep the last step (Denotation) for the next section (5.5.4).

DEFINING THE BUILDING BLOCKS

Here are the various actions on the repository that you'll support. Let's keep it simple for the time being, because the idea is to get an understanding of how free monads help compose DSLs and decouple the algebra from the interpretation. The following listing defines the data types for the actions you support.

Listing 5.6 The building blocks of account repository DSL

```
sealed trait AccountRepoF[+A]
case class Query(no: String) extends AccountRepoF[Account]
case class Store(account: Account) extends AccountRepoF[Unit]
case class Delete(no: String) extends AccountRepoF[Unit]
```

The base trait for actions

Every action expressed as an ADT (pure data)

In the listing, note that you define every action on the repository as a pure data element. The data type for each action defines the algebra without any behavioral semantics attached to it.¹³

THE MAGIC BAND—GET A FREE MONAD

This section discusses the magic mentioned in the previous section. And as is the case with any magic, the public needs to feel only the magical part of it. In this section, you'll get the feel of the magic as the user of the API. The implementation of the API in Scalaz is complex, and you may never need to know the details in order to use a free monad for your domain model. All you need to know is how to make a free monad out of your ADT by using the Free type:

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

This makes AccountRepo a free monad. You must now be wondering why I've been calling the monad *free*. The explanation has its roots in category theory that we won't discuss right now. The article "Free Monoids and Free Monads" by Rúnar Bjarnason goes into more depth in explaining the underlying concepts (<http://blog.higher-order.com/blog/2013/08/20/free-monads-and-free-monoids/>).

Now that you have a monad, the next step is to lift all of your operations into the context of the monad. The result will be a bunch of smart constructors, one for each operation, that you'll be able to compose monadically and build higher-order abstractions.

DEFINING THE MODULE

Let's now lift your actions into the context of the free monad AccountRepo. And this will define your module AccountRepository, as shown in listing 5.7. Unlike the earlier implementation (in section 5.2.1), the module doesn't attach any interpretation to

¹³ This is just what an algebraic data type does.

how the actions will be executed or what values they return. Each action will still return a computation, which is the free monad. And you'll use the `scalaz.Free.liftF` function for the lift.

Listing 5.7 The Module definition for `AccountRepository`—no denotation

```
import scalaz.Free
trait AccountRepository {
    def store(account: Account): AccountRepo[Unit] =
        Free.liftF(Store(account))

    def query(no: String): AccountRepo[Account] =
        Free.liftF(Query(no))

    def delete(no: String): AccountRepo[Unit] =
        Free.liftF>Delete(no))

    def update(no: String, f: Account => Account): AccountRepo[Unit] = for {
        a <- query(no)
        _ <- store(f(a))
    } yield ()

    def updateBalance(no: String, amount: Amount,
        f: (Account, Amount) => Account): AccountRepo[Unit] = for {
        a <- query(no)
        _ <- store(f(a, amount))
    } yield ()
}
```

liftF lifts the `Store` operation into the context of the free monad `AccountRepo`.

COMPOSING THE DSL

Besides the basic operations `store`, `query`, and `delete` that your ADT supplied, you have operations such as `update` and `updateBalance` defined in the module `AccountRepository` in listing 5.7. These are more complex domain behaviors that the monad allows you to build out of the primitive algebra of your ADT. `updateBalance` in listing 5.7 returns a recursive data structure that has the individual components built from the primitive algebra of `Query` and `Store` that you have defined within your ADT.

Using the power of monad composition, you can now build your little DSLs around the APIs that you publish through the `AccountRepository` module:

```
def open(no: String, name: String, openingDate: Option[Date]): AccountRepo[Unit] = for {
    _ <- store(Account(no, name, openingDate.get))
    a <- query(no)
} yield a
```

Creates the account, stores it in the repository, and fetches it for return


```
val close: Account => Account = { _.copy(dateOfClosing = Some(today)) }
```



```
def close(no: String): AccountRepo[Unit] = for {
    _ <- update(no, close)
    a <- query(no)
} yield a
```

Closes the account and fetches the closed account for return

These are account management functions that you can use as part of your domain service APIs. But to see them in action, you need to attach semantics to each of the

operations that you define. And that's what the next section talks about. But before going further, let's have a quick recap of the steps you need to follow to write compositional DSLs using free monads that offer a great degree of separation between the algebra and the interpretation of your model. Figure 5.7 illustrates these steps in detail.

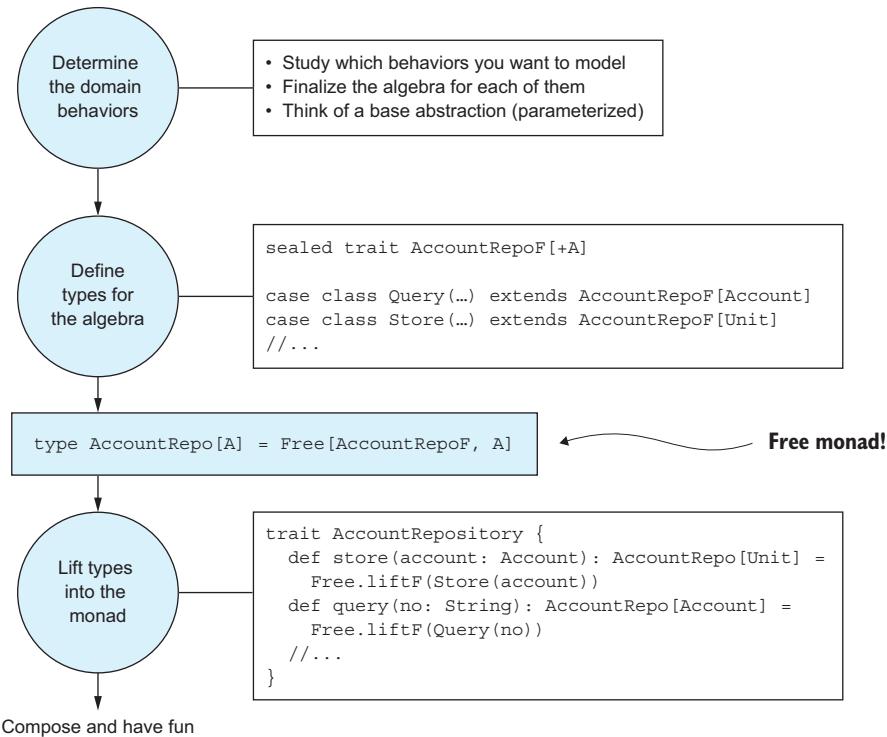


Figure 5.7 Follow these steps to have a free-monad-based implementation of your little DSL. Each step is detailed in the accompanying text, but this diagram serves as a ready reference.

5.5.4 Interpreters for free monads

Now you're down to the last step, which will *execute* your free-monad-based implementation. The interpreter provides the denotation to each of the elements of algebra that your free monad has recursively built within your composed DSL. And the interpretation may also include side effects. Note that you've reached the boundary of your model where you need to interact with the real world. And the real world is side-effecting (you may need to interact with a database or a filesystem, for example, and all of them have side effects built in). The best part is that you have been able to push the side effects to the boundaries of your system, keeping the core algebra pure and functional.

The interpreter is part of the context of your application and it will manipulate the free-monad data structure depending on what you want to do with it. And you may

choose to do multiple things with the same representation of the data. As an example, consider the following little language that you've composed using the preceding ADT for account repository:

```
val account = Account("a-123", "John K")
val comp = for {
    a <- store(account.copy(balance = Balance(1000)))
    q <- query(account.no)
    c <- delete(account.no)
} yield ()()
```

The value that you get back in `comp` has the type `AccountRepo[Unit]`, which is the free monad. You may choose to execute the monad in a certain context or you may choose to have the operations logged as part of an audit trail without any form of execution. You need two interpreters for the two purposes, as you'll see shortly.

As you saw earlier, `AccountRepo` doesn't have any semantics within it; hence it can't be executed directly. The interpreter traverses through the recursive structure of the monad, interprets each component based on what the application wants to do, and collapses the whole computation into the target type. In our example, our target type is a `scalaz.concurrent.Task`¹⁴ and the interpreter will map `AccountRepo` into a `Task`. The following listing shows a sample interpreter for `AccountRepo`.

Listing 5.8 Interpreter for AccountRepository

```
import scala.collection.mutable.{ Map => MMap }
import scalaz._                    ← Basic contract of the
import Scalaz._                   ← interpreter—maps an
import scalaz.concurrent.Task     ← AccountRepo to Task

trait AccountRepoInterpreter {
    def apply[A](action: AccountRepo[A]): Task[A]
}

case class AccountRepoMutableInterpreter() extends AccountRepoInterpreter {
    val table: MMap[String, Account] = MMap.empty[String, Account]

    step is the function that's executed for each node of the AST. →
    fail is a combinator for processing failures in execution. →
    trait AccountRepoF ~> Task = new (AccountRepoF ~> Task) {
        override def apply[A](fa: AccountRepoF[A]): Task[A] = fa match {
            case Query(no) =>
                table.get(no)
                    .map { a => now(a) }           ← now is a combinator on Task that lifts a strict value into a Task[A].
                    .getOrElse {
                        fail(new RuntimeException(s"Account no $no not found"))
                    }
            case Store(account) => now(table += ((account.no, account))).void
            case Delete(no) => now(table -= no).void
        }
    }
}
```

¹⁴ `Task` is similar to `Future` in Scala, with some more computational power. We discuss `Task` in more detail in chapter 6.

```
def apply[A] (action: AccountRepo[A]): Task[A] = action.foldMap(step)
}
```

Runs through all nodes of the AST in the monadic context

Let's look at the salient points in the implementation of the interpreter:

- *Not for production*—The implementation is based on a mutable Map, which we use as the in-memory repository of accounts. Obviously, this isn't something that you'd ideally use in a production setting. But it can be great for testing.
- *Single step*—The step function gets executed for *each* node as you interpret the AST. Each node of the monad is of type AccountRepoF, and the step function defines a mapping from AccountRepoF to a Task. This mapping is special; it's a structure-preserving mapping known as a *natural transformation*¹⁵ and has a special representation in Scalaz (~>). Intuitively, step takes a single node (AccountRepoF), pattern matches it with the elements of our ADT, and creates a Task that when run will execute the desired action.
- *Running the whole*—apply takes the entire AST, traverses it, and invokes step on every node. While in the DSL building phase, you accumulated individual AccountRepoF nodes through flatMap and generated a recursive structure AccountRepo (the Free type). It's the interpreter's apply method that de-structures AccountRepo, gives semantics to each of the nodes, and then flatMaps through the individual Tasks to generate the final Task. Note that Task is also a monad,¹⁶ so you can flatMap through a collection of individual Tasks and get the final Task.



EXERCISE 5.1 EFFECTS IN INTERPRETERS¹⁷

In the free monad implementation that we've discussed, the interpreter in listing 5.8 returns a Task. Task is a monad, and you can interpret it based on the context of your application. Instead of Task, you may want to interpret your free monad in terms of another effect. So why not parameterize your interpreter in terms of the effect that you'd like to produce? Here's an example:

```
trait AccountRepoInterpreter[M[_]] {
  def apply[A] (action: AccountRepo[A]): M[A]
}
```

In this exercise you will explore implementing interpreters in terms of the State monad.¹⁸ Instead of using a mutable Map to store the state, you can abstract the Map as an implementation detail within the State monad. And it can be an immutable Map.

¹⁵ It's not important to know now what *natural transformation* means. The “Free Monoids and Free Monads” post from Rúnar Bjarnason (<http://blog.higher-order.com/blog/2013/08/20/free-monads-and-free-monoids/>) explains the basic concepts and how it relates to free monads.

¹⁶ You can have a look at Scalaz source code to check how the Task monad is implemented.

¹⁷ Jan Vincent Liwanag suggested this exercise. Thanks for the contribution.

¹⁸ I discussed the State monad and stateT in section 4.2.3.

```
object AccountRepoState {
    type AccountMap = Map[String, Account]
    type Err[A] = Error \/ $\backslash$  A
    type AccountState[A] = StateT[Err, AccountMap, A]
}
```

Note the implementation needs to use a Map for storing the account details. You need error handling, which you do using the scalaz. $\backslash\backslash$, and finally have a State monad transformer stacking the error monad with the state.

Implement an AccountRepoInterpreter using AccountState instead of Task. When you pass the free monad and execute the interpreter, it should give you back an AccountState.

5.5.5 Free monads—the takeaways

Like every other pattern in modeling and design, a free monad is a useful one to have as part of your toolkit. As I mentioned at the outset, it's not a frequently used pattern in the Scala community today. But as you must have realized by now, it's powerful. Experts in the community have been advocating for this pattern, and quite a few libraries are coming up that make effective use of this elegant design pattern.

Here are some of the major advantages that this pattern gives to your model:

- *Modularity and testability*—You can modularize your application based on free monads. The algebra can be decoupled from the interpretation in a much stronger way than many other design options. And with modularity, you get the flexibility to swap out implementations and replace them with alternate ones. This is a great advantage when you can plug in mock implementations for testing while going back to the real one in the production environment. So free monads subsume patterns such as dependency injection.
- *Purity*—You can keep things pure and algebraic even when composing the whole structure of the DSL. The free monad gave you the whole abstract syntax tree, much as you'd get from Lisp macros. And the whole thing is typed and checked during compile time. So you have the structure that you can reuse in other contexts and reason about mathematically. The separation between the algebra and the interpretation is much more explicit.
- *Scalability*—In Scala, which doesn't support generic tail calls, the free monad implementation lets you scale your DSL to arbitrary levels of complexity without the fear of blowing your stack. The scalaz.Free implementation uses trampolining that trades heap space for a stack.¹⁹

¹⁹ See “Stackless Scala with Free Monads” by Runar Bjarnason (<http://blog.higher-order.com/assets/trampolines.pdf>).

5.6 Summary

This chapter presents one of the fundamental topics in software engineering in general and domain modeling in particular. Unless you learn how to write modular software, your model turns into a monolithic piece of code. And that's difficult to refactor, extend, and maintain in the long run. In this chapter, you saw how to decompose your model into modules in Scala. The main takeaways of this chapter are as follows:

- *What is modularization?*—You learned what constitutes a module: the algebra and implementation. You also saw how to publish the algebra of your module and at the same time protect its implementation.
- *Anatomy of a module*—You took a module fragment from the domain of personal banking and learned how to decompose it into modules. You designed the algebra, looked at the domain behaviors, and saw how to make the algebra compositional. You also looked at the implementation of the algebra and how to decouple it from the algebra.
- *Compositionality*—The domain behaviors that you define within module algebra need to compose so that you can write compositional code using the algebra. You can achieve this compositionality at the algebra level so that you can afford to commit to the implementation only at the client application level.
- *Physical organization of modules*—You can organize modules into packages as physical artifacts and publish them selectively so that only the artifacts that you need get exposed to the client.
- *The Type Class pattern*—You can implement polymorphism on specific behaviors across a set of unrelated abstractions. You saw the advantages that it offers over subtype polymorphism. This pattern allows you to add behaviors to existing classes on a post hoc basis.
- *Bounded context*—A bounded context provides a larger granularity of modularization than a simple module. You saw how to implement bounded contexts and how to manage communication between multiple bounded contexts.
- *Free monads*—This is the last topic you looked at. It's an advanced pattern of modularization that you can skip on first reading. But it provides a lot of power in separating the structure of your computation from the interpretation. The core idea is that you define your computation units as pure data types over a closed ADT. You can then use the Free data type to get a monad, on which you can structure your DSL. You get this entire structure as an AST, which you can interpret later.

Being reactive



This chapter covers

- Introduction to reactive domain models
- Reactive API design using futures and promises
- Reactive communication between bounded contexts using asynchronous messaging
- The stream model with example implementations from Akka Streams
- The actor model of communication

So far you've learned about the virtues of functional programming and how to apply these principles when you design a domain model. This chapter starts the discussion of how to make your functional domain model *reactive*. You want your model to be responsive to users, elastic for scalability, resilient to failures, and decoupled in time and space using message-driven architectures.¹ This chapter covers various ways of being reactive with domain models built using Scala. You'll take a step up from the low-level concurrency primitives such as threads and semaphores and use higher-order abstractions that Scala offers. These include designing

¹ The Reactive Manifesto, www.reactivemanifesto.org

APIs using futures and promises, implementing loosely coupled architectures with asynchronous messaging, building stream pipelines, and finally using the actor model that also provides fault tolerance and redundancy.

Figure 6.1 shows a schematic of this chapter's topics. This guide will help you selectively choose your topics as you sail through the chapter.

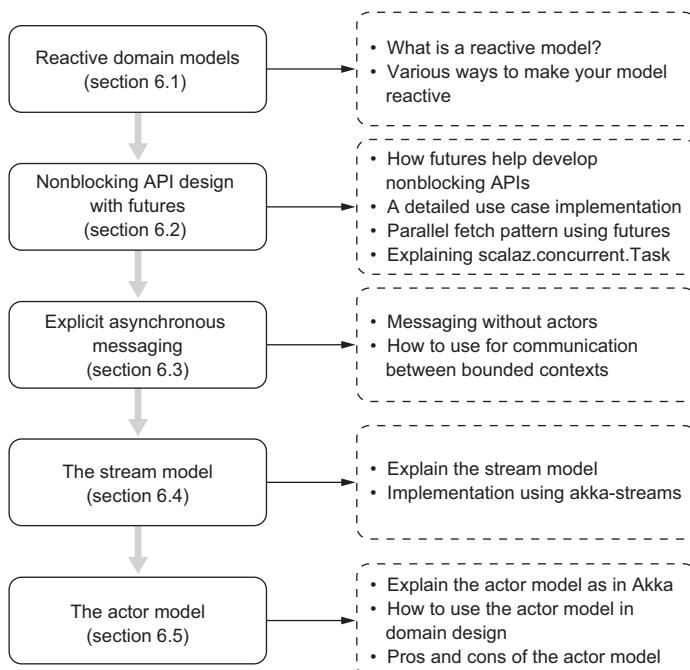


Figure 6.1 The progression of this chapter's sections

At the end of the chapter, you'll understand how to use proper abstractions to make your domain model responsive to users and resilient to failures.

6.1 **Reactive domain models**

You've already seen the basic principles of reactive models in chapter 1. Figure 1.10 in that chapter presented the core attributes that a reactive model should have. But that discussion was mainly driven by generic principles of system design. This section reconciles those thoughts with the functional domain model architecture discussed in the last few chapters.

Based on our discussions of models and bounded contexts, figure 6.2 depicts the overall structure that any domain model of moderate complexity should have. And this certainly includes the model we've been discussing so far: the domain of personal and investment banking.

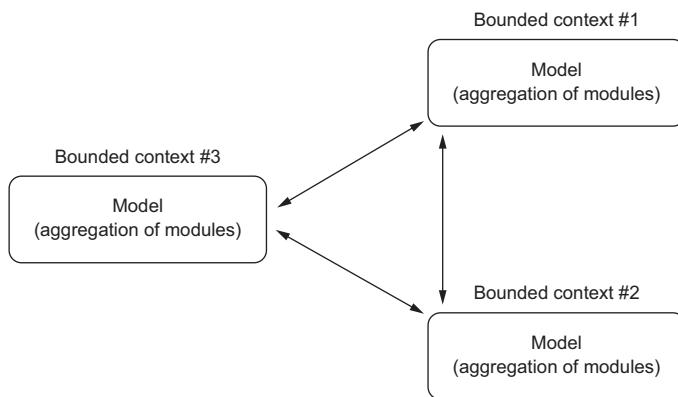


Figure 6.2 Overall architecture of domain models. Each nontrivial model has many models within it, each architected as a separate bounded context.

A domain model of any significance has multiple distinct models within it, each with a set of different constraints, distinct business rules, and specific ubiquitous language describing the vocabulary of the model. So what exactly do I mean when I say that you want your model to be reactive?

The ultimate goal is to make the system as a whole responsive so that the user feels comfortable with its overall response. As discussed in chapter 1, this means that the system must respond to the user's request within a specific time delay, also called a *bounded latency*. And remember that bounding the latency of a system also includes failures. A system that gets stuck on failures obviously isn't responsive. Chapter 1 covered various strategies for handling failures.

To make a model reactive, you need to ensure that all of its constituent models are also reactive. All principles of reactive architecture that hold for one model must be equally applicable to all the constituent models of the system. A full system is as reactive as its least reactive component.

One common question that arises is what components need to be reactive within a model? Is it true that *all* functions need to be nonblocking? Or *all* objects need to be actors that implement model behaviors using asynchronous messaging? Some schools of thought espouse this principle of *actors all-the-way-down*. But this design principle has some strong disadvantages that we'll discuss in this chapter. For the moment, let's assume that you'll apply your judgment when deciding which functions need to be nonblocking, and which (if any) need to be implemented in terms of asynchronous messaging. But one thing is true: You need to consider being reactive as part of your overall architecture. It's not something that you can bolt on in the later phases of implementation as an afterthought.

This section presents an overview of the tools at your disposal for implementing the reactive paradigms in your domain model. Not all of them may be applicable for

your use case, but it's always good to have an overall idea. At least then you can justify the choices you make. In the following sections, you'll dig deeper into each of them and I'll discuss various implementation techniques.

Here's a list of the techniques that you can use to make your domain model reactive:

- *Futures and promises*—Using futures, you can design asynchronous, nonblocking APIs for your model. Instead of blocking on the future, you can compose multiple futures by using higher-order functions and generate one final end result. Using futures shapes the way you design your APIs. You'll see how to improve some of the APIs that you designed in chapter 5.
- *Explicit asynchronous messaging*—When you have different bounded contexts interacting, it's always recommended to have these interactions loosely coupled. For a more detailed discussion, see section 5.4.2 in chapter 5. Asynchronous messaging is the ideal tool for modeling such interactions, because it gives a nice decoupling in space and time. Design these APIs around messaging so that you don't block while communicating between two disjointed models of your domain.
- *The actor model*—This is one of the primary architectural constructs that help you design your model based on asynchronous, event-driven, and nonblocking communication. The sender sends messages based on the fire-and-forget style, and the receiver's mailbox gets them for further processing. The primary takeaway of this model is that you don't have to handle concurrency within your code. An actor allows a single thread of execution within itself. You achieve concurrency by creating a huge number of actors, which are lightweight abstractions of computation. Some actor implementations such as Akka (<http://akka.io>) also offer features such as fault tolerance, location transparency, and distribution.
- *The stream model*—You can use this model when you need to work with one or more streams, where data arrives, possibly from an endless source, and you need to compose for effectful processing. These are called *reactive streams* (<http://www.reactive-streams.org>), of which there are a few implementations, including Akka Streams (<http://doc.akka.io/docs/akka/2.4.4/scala/stream/index.html>) and scalaz-stream (<https://github.com/scalaz/scalaz-stream>). You'll look at Akka Streams in this chapter.

Each technique has a different level of granularity and affects your model design in different ways. Some are more invasive than others and need careful and up-front planning. Some are related to the API design and have an *inside-out* effect on your model. As an example, when you change your APIs from blocking function calls to nonblocking future-based ones, the impact originates at the API level and spreads out into application layers. Others, such as supervision and fault tolerance, are *outside-in* and need to be injected from the periphery of your model. Computation models such as the actor model tend to have an overhauling impact on your design; all interactions are through messaging instead of function calls. Not much of typed computation goes

on underneath the fire-and-forget message-processing model of actors. Hence it becomes difficult to reason about your code when using this model as the end-user interaction point. You'll explore these and look at the scenarios for which this model makes the most sense within your domain model. You'll start with the most common way of making your model reactive: designing APIs with futures and promises.

6.2 Nonblocking API design with futures

Remember that one of the core tenets of being reactive is to ensure that your design doesn't create any contention or central bottlenecks that can hamper the progress of the system. Imagine that your domain service publishes APIs that block calls to underlying databases and block the central thread of user interaction with an unbounded latency. This is a clear indication that your API design isn't proper; blocking kills the flow and leads to a bad user experience.

This section takes an example from the module APIs designed in chapter 5 and uses futures to make them nonblocking and asynchronous. As mentioned earlier in this chapter, you'll make a call as to which APIs you need to make nonblocking instead of being blown away in the fandom of making the whole model asynchronous. At the end of this discussion, you'll see why making these APIs nonblocking is a reasonable strategy.

As an example, let's consider the algebra of the module discussed in section 5.2.1. Here's the algebra once again in the following listing.

Listing 6.1 Algebra of the module AccountService

```
trait AccountService[Account, Amount, Balance] {
    type Valid[A] = NonEmptyList[String] \/\ A
    type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

    def open(no: String, name: String, rate: Option[BigDecimal], ←
            openingDate: Option[Date], accountType: AccountType): ←
        AccountOperation[Account]

    def close(no: String, closeDate: Option[Date]): AccountOperation[Account]
    def debit(no: String, amount: Amount): AccountOperation[Account]
    def credit(no: String, amount: Amount): AccountOperation[Account]
    def balance(no: String): AccountOperation[Balance]

    def transfer(from: String, to: String, amount: Amount) = //..
}
```

The code listing is annotated with two callout boxes:

- A box on the right side, labeled "Module definition—domain-friendly name and parameterized on types", points to the trait definition: `trait AccountService[Account, Amount, Balance] {`.
- A box on the right side, labeled "Operation definition with domain-friendly name", points to the `open` method definition: `def open(no: String, name: String, rate: Option[BigDecimal], ←`.

Each module API returns a Kleisli, which as you saw earlier nicely composes while implementing larger functionalities. When you designed this module, you ignored one important aspect: the interaction with the underlying `AccountRepository`² and

² Typically, it will be an enterprise database.

the possibility that it may lead to unbounded latency at the application level. If these operations take a long time, the main thread of execution will be blocked and will lead to poor user experience. This is one use case where we need to make your APIs *elastic* enough so that the perceived response to the user isn't affected by the current load on the system. And as discussed in chapter 1, this is the essence of future-based computing: Make the operations nonblocking by returning `Future` as the computation. How do you fix this?

6.2.1 Asynchrony as a stackable effect

You need to change the APIs and their implementations. At the same time, you want to express your reactive intent through the types of your APIs. In the current algebra, the Kleisli helps curry `AccountRepository` and effectful function application. You'd like to have it the same way in the new version as well. The crucial part is the disjunction, `scalaz. \ /`, which is a *sum type* that returns the value or the error in computation.³ You need to plug in your `Future` somewhere before you compute the disjunction.

This is an exercise in combining two effects, disjunction (`\ /`) and `Future`, layered in a specific manner so that `Future` is available immediately, and the disjunction is available only when the computation is complete. Fortunately, both effects are monads, and you can use the standard technique of layering them by using *monad transformers*. You'll look at the implementation, but first let's take a brief look at monad transformers.

MONAD TRANSFORMERS

In chapter 4, you saw that when you do monadic binds (`flatMap` in Scala), the execution of one step in the chain depends on the success or failure of the last one. Hence a monadic effect doesn't preserve the shape of the computation. For a refresher, see section 4.2.3. Because of this, we say that *monads don't compose* in the generic way that applicatives do.

To create a composite monad out of many, you can use monad transformers. This technique stacks monads together, giving you a transformed monad. You can then use comprehension on the composed monad and extract the value out of it. This isn't a generic technique and works for only a select set of monads.

Consider the following example, which models the return type of a query of domain objects from a repository:

```
type Response[A] = String \ / Option[A]
```

Here `Response` is a monad, because `\ /` is a monad in Scalaz. With an ordinary right-biased `Either` type (which `\ /` is), you get the value from the right projection. But here the value at the right is an `Option`. To reach the count that the query returns, you have

³ I discussed sum types and product types in section 2.4.1 in chapter 2.

to process another monad within $\backslash\!/\!$. So how will you extract the final value of `count` from a `Response`? Here's a first attempt that peels off the monadic layers one by one:

```
val count: Response[Int] = some(10).right
for {
    maybeCount <- count
} yield {
    for {
        c <- maybeCount
    } yield c
}
```

The problem is that this approach doesn't scale. Note the second level of for-comprehension that's nested within the first. With more effects stacked within `Response`, you'll have more nesting of the comprehension, and your code will threaten to fall off the right margin of your indentation. It would be ideal if you could take the $\backslash\!/\!$ monad and add the effect of handling optionality to the right value (that is, layering an `Option` on top of $\backslash\!/\!$ so that you can treat the combination as a single monad). It's like combining the two monads into one. This is exactly what a monad transformer does. Here's our example that incorporates this new layering:

```
import scalaz.{ \!/, OptionT }
type Error[A] = String \!/ A
type Response[A] = OptionT[Error, A]
```

Here `OptionT` is a monad transformer that gives you a single monad that adds `Option` as a layer on top of an underlying monad, which in this case is $\backslash\!/\!$. Now you can reach into the value of `count` by using a single for-comprehension. Here's the same code using `OptionT`:

```
import scalaz.syntax.monad._
val count: Response[Int] = 10.point[Response]
for {
    c <- count
} yield ()
```

From the user point of view, this looks much cleaner because you've been able to get rid of the extra nested steps within the for-comprehension. Let's use this technique in your implementation of stacking `Future` and `Either` into one single monad and implement asynchronous effects, as discussed at the beginning of section 6.2.1.

STACKING FUTURE AND EITHER

The current model uses the transformer `EitherT`, which gives you the exact layering that you need here. This is an important concept to internalize; do you see the problem of using `Future[NonEmptyList[String]] \!/ A` instead? It's the same issue discussed with `OptionT`. You need to jump through additional indirections with nested for-comprehensions in peeling off the multiple monadic layers of your computation.

In contrast, the monad transformer `EitherT` gives you a single monad by stacking the effect of `Future` within an `Either`. The basic idea to take away from this modeling technique is *to make your type as close as possible to the effect that you're trying to implement*.

When you use types as an intrinsic part of your model definitions, the layering of effects is mostly an additional type plugged into an already existing stack. Here you add `Future` as an *effect* to model reactive APIs and add a layer to the monad transformer. Proper type definitions can localize such changes only to a single type alias that defines the transformer. You need only to change the alias `Valid` to type `Valid[A] = EitherT[Future, NonEmptyList[String], A]`, and everything else in the algebra remains the same as in listing 5.1 in chapter 5. Figure 6.3 illustrates this technique for our example.

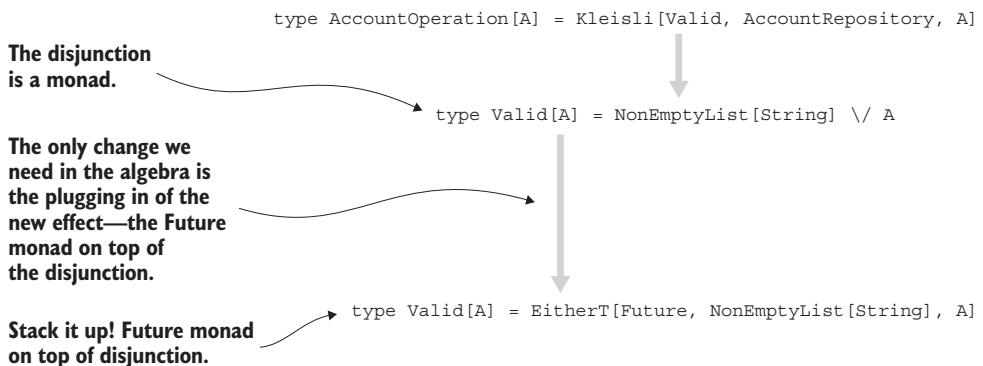


Figure 6.3 Type-driven effect composition: you treat asynchronous and nonblocking as an effect and model it through a monad transformer. The only change in the algebra is in the type alias `Valid`. The rest of the algebra remains unchanged.

6.2.2 Monad transformer-based implementation

Let's also look at the change that needs to be made in the implementation of the module API. As shown in the algebra here, you need to stack the two effects by using the transformer, and the core domain logic remains the same as in listing 5.2 in chapter 5. The following listing shows an example.

Listing 6.2 Interpreter for AccountService

```

package domain
package service
package interpreter
    ↪ Imports as in
    ↪ listing 5.2

class AccountServiceInterpreter extends
  AccountService[Account, Amount, Balance] {

  def open(no: String,
          name: String,
          rate: Option[BigDecimal],

```

```

openingDate: Option[Date],
accountType: AccountType) =
kleisli[Valid, AccountRepository, Account] {
  (repo: AccountRepository) =>
    EitherT {
      Future {
        ...
      }
    }
  //... other operations
}

```

Stacking Either with Future using monad transformer EitherT

Same as listing 5.2

You must now be wondering about the impacts that a well-designed type system can have on your model. You've introduced a whole new effect into your API algebra and implementation *just by composing with another type, without any change in the core domain logic*. This is the essence of typed functional programming. Types model effects functionally, and you can stack them up in the proper order whenever you need to.

Why monad transformers?

As discussed in chapter 2, an effectful computation adds capabilities to an existing type. You can stack these capabilities to make your model more powerful. For example, in the preceding use case, you needed to combine the effects of asynchronous execution along with error handling as a monadic effect. (For a refresher on monadic effects, see section 4.2.3 of chapter 4). But one problem in composing monadic effects is that *monads don't compose in general*. Some specific monads may compose, but monads aren't closed under composition.

Enter monad transformers. With transformers, you can take a monad and add the effect of another monad on top of it so that the result is a monad that combines the effect of both of them. The order of stacking matters. You specify the order when you create a monad transformer of the appropriate type. The benefit is that you can now compose the effects any way you wish and still use a single for-comprehension to navigate through the stack of effects. Have a look at the running example that we've used in this section for discussion.



EXERCISE 6.1 HANDLING EFFECTS (ALGEBRAICALLY)

In chapter 4, section 4.4 described a complete example of how to evolve an API by using the algebra of types and patterns. You finally arrived at the implementation of a `tradeGeneration` function that generates trades from client orders and market executions. And you did that without any idea about the implementation of the underlying types. Listing 4.10 shows the final function that you implemented.

You'll notice that each function that you composed to form the implementation of `tradeGeneration` doesn't handle errors that may arise in the course of its

execution. For example, `allocate(as: List[Account])` takes `Execution` and generates a `List` of `Trade`. But what happens if there's an exception? The function doesn't return to the caller any information regarding the exceptions that may occur during execution of `allocate`.

Each function already handles an effect in the form of `List`. Given a list of client accounts and an execution, `allocate` returns a `List` of trades. If you have to handle error reporting, one way to do that is to handle exceptions as effects that need to be stacked along with the already existing effect (`List`). You can't throw exceptions (a strict no-no in FP land) or you violate referential transparency.

In this exercise, implement the entire use case so that every function handles exceptions and reports proper error messages to the caller. The control flows will automatically have to be adjusted. If allocation fails for any execution, you don't generate any trade. If the execution fails for any order, you don't generate any execution.

Hint: Use a suitable representation for handling exceptions as effects and stack it with the `List` effect using monad transformers. Explore Scalaz for already-existing abstractions.

6.2.3 Reducing latency with parallel fetch—a reactive pattern

In the example in sections 6.2.1 and 6.2.2, say you compose the Future-based operations of `AccountService` as follows:

```
for {
    _ <- open(..)
    _ <- credit(..)
    d <- debit(..)
} yield d
```

Then the composed operation is a `Future`, but the individual operations making up the for-comprehension will be sequential. Here you gain from the fact that the composed operation doesn't block the main thread of execution, and you can retrieve the results when it completes. And this is indeed the valid way to execute the preceding use case; making the individual operations execute in parallel will obviously lead to unwanted nondeterminism.

But in some use cases you can benefit by making the granular operations execute in parallel. Chapter 1 described one such use case in a discussion on event-driven programming using `Futures`. In this section, you'll consider the implementation in detail. You'll skim through the design here, but you can get the full implementation in the code repository for the book.

Let's say our bank offers a portfolio service for its customers. You create a simple domain model for the customer portfolio and then design module APIs for reporting the composite portfolio. The following sidebar gives a brief overview of what to expect as part of your portfolio statement.

Portfolio service—what's a portfolio?

A customer portfolio refers to the details of a customer's holdings with the bank. This includes all currency holdings as well as equity, fixed income, and other instrument holdings. The portfolio statement consists of the following details for each kind of holding:

- ISIN code, the international code for the instrument
- Holding amount
- Current market value

Customers can get a quick snapshot of their overall financial position with the bank from their portfolio statement.

You have a simple model for a customer account and an abstraction for modeling the portfolio entity. Listings 6.3 through 6.5 describe the domain model for Portfolio, Instrument, and Account.⁴ Instrument is an entity that forms the basis of all financial transactions. In case you're not familiar with financial instruments, the following sidebar gives a brief overview.

Financial instruments

A financial *instrument* is a unit of capital with specific characteristics that can also be traded in the market. Trading of instruments in the market allows efficient flow of capital across investors.

This chapter covers the following types of instruments:

- Currency (CCY), which is the simplest instrument in the market. Currencies can have different codes such as USD (U.S. dollar) or EUR (euro).
- Equity (EQ), which is a stock that you can own.
- Fixed income (FI), which usually comprises bonds that provide fixed periodic payments and eventually return the principal at maturity.

Listing 6.3 The Portfolio domain model

```
package model
import java.util.Date
import common._

case class Balance(ins: Instrument, holding: Amount,
  marketValue: Amount)

sealed trait Portfolio {
  def accountNo: String
```

Balance is a value object. For optimization, you can keep an instrument ID here and look up a repository for the instrument.

⁴ All domain models are simplified for easy explanation.

```

def asOf: Date
def items: Seq[Balance]
def totalMarketValue: Amount =
    items.foldLeft(BigDecimal(0d)) { (acc, i) => acc + i.marketValue }
}

case class CustomerPortfolio(accountNo: String, asOf: Date,
    items: Seq[Balance]) extends Portfolio

```

totalMarketValue is a computed field.

Listing 6.4 The Instrument domain model

```

package model

import java.util.Date
import common._

sealed trait InstrumentType

case object CCY extends InstrumentType
case object EQ extends InstrumentType
case object FI extends InstrumentType

sealed trait Instrument {
    def instrumentType: InstrumentType
}

case class Equity(isin: String, name: String, issueDate: Date,
    faceValue: Amount) extends Instrument {
    final val instrumentType = EQ
}

case class FixedIncome(isin: String, name: String,
    issueDate: Date, maturityDate: Option[Date],
    nominal: Amount) extends Instrument {
    final val instrumentType = FI
}

case class Currency(isin: String) extends Instrument {
    final val instrumentType = CCY
}

```

You consider these three instrument types here.

A ridiculously simplistic model for Instrument

Listing 6.5 The Account domain model

```

package model

import java.util.{ Date, Calendar }
import scalaz._
import Scalaz._

object common {
    type Amount = BigDecimal
    def today = Calendar.getInstance.getTime
}
import common._

case class Account(no: String, name: String,
    dateOfOpen: Option[Date] = today.some, dateOfClose: Option[Date])

```

In this section, you'll explore the use case of having a service that generates the full portfolio of a customer. This full portfolio includes currency, equity, and fixed income. Typically, because these are different types of instruments, the portfolio information is fetched from different systems. Therefore, they can be fetched in parallel and finally composed together to form the complete customer portfolio. Figure 6.4 illustrates the scenario.

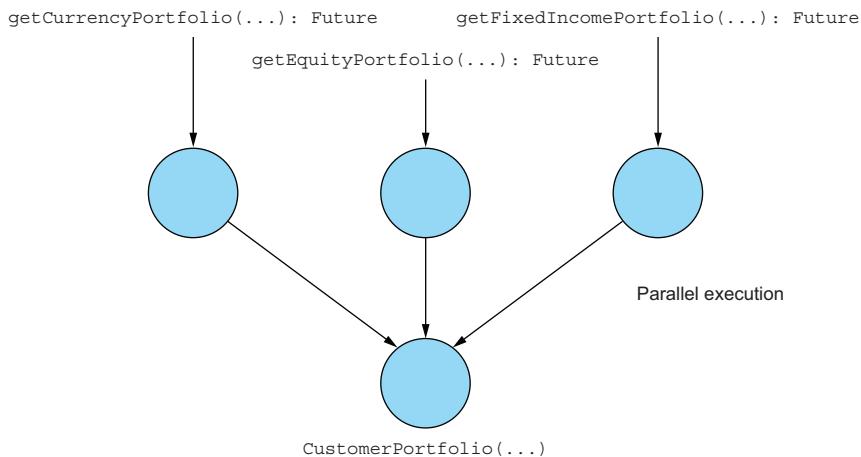


Figure 6.4 Parallel execution of portfolio-fetching methods to compose the full portfolio

And the following listing shows the service contract for the module named `PortfolioService`.

Listing 6.6 Portfolio reporting service

```

trait PortfolioService {
  type PFOperation[A] = Kleisli[Future, AccountRepository, Seq[A]]
  def getCurrencyPortfolio(no: String, asOf: Date): PFOperation[Balance]
  def getEquityPortfolio(no: String, asOf: Date): PFOperation[Balance]
  def getFixedIncomePortfolio(no: String, asOf: Date): PFOperation[Balance]
}

```

As earlier, you use Kleisli
to inject the repository.

Every method that returns a specific portfolio fetches the list of balances for all instruments of that particular type. For example, a customer may have an equity balance for many equities, and the `getEquityPortfolio` method gives you a list of all balances.

This chapter doesn't cover the implementation of this service; the full code is in the online code repository for the book. Let's focus on how to use these services to build a nonblocking, reactive portfolio computation function. The following listing gives a sample implementation.

Listing 6.7 Building the complete portfolio

```

import PortfolioService._

val accountNo = //...
val asOf = //...

val ccyPF: Future[Seq[Balance]] =
  getCurrencyPortfolio(accountNo, asOf) (AccountRepositoryInMemory)
val eqtPF: Future[Seq[Balance]] =
  getEquityPortfolio(accountNo, asOf) (AccountRepositoryInMemory)
val fixPF: Future[Seq[Balance]] =
  getFixedIncomePortfolio(accountNo, asOf) (AccountRepositoryInMemory)

val portfolio: Future[Portfolio] = for {
  c <- ccyPF
  e <- eqtPF
  f <- fixPF
} yield CustomerPortfolio(accountNo, asOf, c ++ e ++ f)

```

Gets the currency portfolio

Gets the equity portfolio

Gets the fixed-income portfolio

Makes the composite portfolio. Each of ccyPF, eqtPF, and fixPF is a Future[Seq[Balance]]. Hence within the for-comprehension, c, e, and f are Seq, and you can use the concatenation operator ++ to form the composite portfolio.

You first invoke the separate methods for building specific portfolios. Each returns a Future and hence can potentially be executed in parallel. Then you bind them through a for-comprehension and build the final portfolio. The code has all the type annotations for explanation; in your model, you can skip them. But please convince yourself that moving the creation of the vals ①, ②, ③ within the for-comprehension no longer makes the implementation parallel and nonblocking.

6.2.4 Using scalaz.concurrent.Task as the reactive construct

Chapter 5 presented another implementation of modules and their composition, the one based on free monads. (To refresh your memory, see section 5.5.2.) Free monads help you build an abstract syntax tree of recursive data types without applying any interpretation to them. You're completely free to interpret the tree as you wish.

The free monad is an encoding of the computation as an abstract syntax tree. To make a reactive implementation out of it, you need to make the interpreter reactive. That's exactly what you did in `AccountRepoInterpreter` in section 5.5.4. But instead of `Future`, you used another construct, `scalaz.concurrent.Task`:

```

import scalaz.concurrent.Task
trait AccountRepoInterpreter {
  def apply[A](action: AccountRepo[A]): Task[A]
}

```

`Task` abstracts asynchronous execution as an effect through a type class, Nondeterminism. The following sidebar provides more information on the `Task` abstraction and how it differs from `Future`. Because our interpreter returns a `Task`, the implementation is already reactive!

For our portfolio example, you can have the methods of `PortfolioService` return a `Task` instead of a `Future`. And then the application code for composing the complete portfolio becomes much simpler. The following listing illustrates this.

Listing 6.8 Using Task for PortfolioService

```
import PortfolioService._

val accountNo = "a-123"
val asOf = today

val ccyPF: Task[Seq[Balance]] =
  getCurrencyPortfolio(accountNo, asOf) (AccountRepositoryInMemory)
val eqtPF: Task[Seq[Balance]] =
  getEquityPortfolio(accountNo, asOf) (AccountRepositoryInMemory)
val fixPF: Task[Seq[Balance]] =
  getFixedIncomePortfolio(accountNo, asOf) (AccountRepositoryInMemory)

val r = Task.gatherUnordered(Seq(ccyPF, eqtPF, fixPF))      ←
val portfolio = CustomerPortfolio(accountNo, asOf,
  r.run.foldLeft(List.empty[Balance]) (_ ++ _))
```

gatherUnordered
executes tasks in
parallel.

scala.concurrent.Future & scalaz.concurrent.Task

As you've seen, `Future` (from the Scala standard library) and `Task` (from `scalaz.concurrent`) offer ways to make your model reactive. Both help you run your APIs asynchronously and in a nonblocking fashion. But the two abstractions have fundamental differences. When you choose which one to use, you need to apply your own judgment:

- `Future` is an abstraction from the standard library, so you don't have to depend on any third-party library for using `Future`.
- `Future` models a running computation; it abstracts how you *build up the computation* and how you *run* it in a single step. This is somewhat unclean as far as the engineering aspects of API design are concerned. `Task`, on the other hand, has a clear separation between how you form the *description of the computation* and its *execution*. You can control the degree of nondeterminism through an explicitly supplied context, a type class named `Nondeterminism`.⁵ Intuitively, this context allows you to supply the exact behavior of nondeterminism that you'd like to model when the task executes. `Task` is a monad. So when you execute a `Task`, the default strategy is to *bind*, just as a monad does. But with an explicitly specified execution context like `Nondeterminism`, you can change the execution strategy. For example, you can say that you have so many things to execute—instead of doing a sequence of binds, executing them in parallel, and returning the first completed one.

⁵ Have a look at Scalaz source code for more details on `Nondeterminism`.

- Future-based APIs can be made monadic by providing a `Monad` instance of `Future`. This is typically the strategy you use when making reactive APIs. But as you saw in section 6.2.1, this requires using monad transformers if you want to stack a `Future` on top of existing monadic APIs. Monad transformers are not too intuitive, especially in Scala, and are known to produce hard-to-understand code structure. `Task` offers lots of combinators that don't necessitate monad transformers. Hence design and use of Task-based reactive APIs are often simpler.
- `Task` is a trampolined abstraction, which means it can work on constant stack space. This also becomes an important advantage when dealing with interleaved concurrency structures such as cooperative coroutines. Have a look at "Stackless Scala with Free Monads" by Runar Bjarnason (<http://blog.higher-order.com/assets/trampolines.pdf>) for more on how trampolines can help design deeply nested constructs without using unbounded stack space.

The primary takeaways from this discussion are the following:

- *Reactive inside out*—The reactive traits discussed here relate to the design and implementation of APIs that flow *outward* from the core of your model toward the consumer application.
- *Layering of types*—The implementation technique that we discussed is completely noninvasive. Taking advantage of monad transformers, you could plug in the reactive nature by layering a `Future` as an effect on top of the already existing monad.
- *Asynchronous and nonblocking*—By making the APIs nonblocking, the main thread remains free for other activities. You're not stuck waiting for a blocking API call returning with a bunch of data from a database hosted in a faraway data center.
- *Freedom implies more goodness*—If you have a free-monad-based architecture, the free monad is completely decoupled from any effect you add. All you need to do is change the interpreter.

The online code repository has the complete source code of the reactive version of the API and the implementation.



EXERCISE 6.2 FUTURE AND SCALAZ.CONCURRENT.TASK

The `AccountService` module in listing 6.1 uses monad transformers to stack `Future` into your already monadic API. Reimplement the service using `scalaz.concurrent.Task` in place of `Future`. Refer to the discussion in the preceding sidebar that compares `Future` with `Task`.

Hint: You don't need to use monad transformers. The API becomes simpler than the one based on `Future`.

6.3 **Explicit asynchronous messaging**

When you talk about reactive systems, the first item that comes to mind is how they address coupling issues between the various parts of the system. You'll often hear people arguing that in order to keep a system reactive, you need to make components as loosely coupled as possible. What does this mean? How do you know which of the components need to be loosely coupled? And how exactly can you ensure this loose coupling?

One important concept that you need to understand in this context is that of an *asynchronous boundary*. When two entities that are decoupled in space and time need to communicate with each other, that communication has to be asynchronous. Synchronous calls will either block for a long time or fail because there's no agreed-upon protocol between the two parties regarding the communication. This is an example of an asynchronous boundary, where you need to pass information between two parties asynchronously. Many asynchronous boundaries exist in the real world. You can have asynchronous boundaries between the following:

- Applications or bounded contexts
- Threads
- Actors
- CPUs with multiple cores
- Network nodes

Even though you have two bounded contexts collocated within the same machine, you may need to consider handling data movement across asynchronous boundaries. This section presents use cases for designing domain models that use asynchronous messaging. This section doesn't cover the actor model, as it's explained in a separate section.

The classic use case for asynchronous messaging is the modeling of the communication between multiple bounded contexts. A bounded context is typically a separate application running within the realms of its own space and time. But bounded contexts also need to interact between themselves, though not on a regular basis. Consider the example in chapter 5 with two bounded contexts: one for Account Management that deals with online requests for managing customer accounts, and the other for Reporting and Analytics that's responsible for generating reports and MIS information for all accounts in the system.

You can't have synchronous communication between the two contexts for the following reasons:

- They may not be collocated.
- They may be modeled as services that run under different temporal constraints (for example, reporting may run only as a nightly job, whereas account management may be a 24/7 service).
- They will have different data models and ubiquitous languages with different vocabulary (for example, Account may be a valid artifact in both contexts, but with a different set of attributes).

Figure 6.5 describes the use case with the two bounded contexts. In this scenario, the Account Management context generates events/messages that contain all the information that the consumer context needs. But Reporting is the *conformist* context, which means that it has to conform to the message format/protocol that Account Management generates.⁶

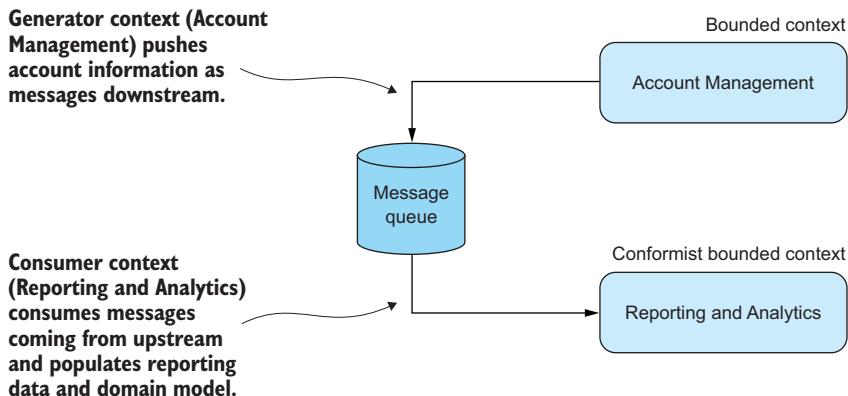


Figure 6.5 Bounded contexts separated by time and space. They communicate through asynchronous messaging.

Messaging takes place through a message queue. The generator context (Account Management) pushes messages into the queue, and the consumer context (Reporting) pulls at its own convenience. The message queue can be modeled using any of the suitable technologies such as Kafka (<http://kafka.apache.org>) or RabbitMQ (www.rabbitmq.com) or any such store that offers the semantics of a queue.

The primary takeaways on using explicit asynchronous messaging are as follows:

- *Asynchronous boundaries*—Use this strategy to pass information across asynchronous boundaries. And remember, asynchronous boundaries can occur even within the same machine across threads or even across CPU cores.
- *Decoupled in time and space*—This model of programming offers decoupling in time and space. Use this strategy to communicate between systems that are loosely connected (for example, communication between bounded contexts).

6.4 The stream model

Communication between asynchronous boundaries by using explicit messaging is convenient and often the most suitable model of information exchange, as you saw in the preceding section. This is convenient when the communication is sparse and

⁶ See *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley Professional, 2003).

systems are spatially and temporally decoupled. But often you'll find that one component of your model generates a stream of data that needs to be processed and transformed in various ways before being consumed by multiple downstream components. This section presents some approaches for handling the stream model of data interaction to help keep your system reactive.

Let's start with some of the specific challenges that this model offers from a design perspective:

- *Continuous flow of information*—In many cases, streams are infinitely generative (for example, stock indexes, market prices, exchange rates, and time series data from monitoring systems that flow into your system on a continuous basis). How do you process them in a manner that doesn't create an overwhelming load on your system?
- *The solitary look*—For stream data, you get to see the data once. The processing has to be online, and you can't drop any piece of data from the stream.
- *Storage issues*—When you have a continuous flow of data, you can't assume that you can store all of it and then do processing. Again, processing has to be online and with minimum latency.
- *Back-pressure handling*—In most cases, you'll see that the consumer of data is not fast enough to cope with the rate at which the producer throws stuff at it. How do you manage this mismatch?

Keeping in mind these issues, here are some of the suggested pathways to address them. Not every stream-processing framework will implement all of these features. But, as a model designer, it's always good to have the whole perspective. Here are some attributes that a scalable, stream-processing API should have:

- *Asynchronous*—The process has to be asynchronous in order to ensure optimal utilization of computing resources across collaborating network nodes or even among multiple CPU cores within a single machine.
- *Nonblocking*—This is required to ensure bounded latency, similar to what we discussed in the context of future-based APIs in section 6.2.
- *Elastic for handling back pressure*—This is needed when producers push data toward consumers and the consumer may be processing at a slower rate than what the producer is generating.
- *Compositional*—This is a quality that we demand all APIs should have. This will lead to a better programming model.

6.4.1 A sample use case

Instead of going through more theory behind designing APIs for the stream model, let's consider a sample use case from our domain and see how to model it by using one of the stream-processing frameworks.

In this use case, you'll consider downstream processing of banking transactions. You have a source that receives a stream of banking transactions from one or multiple

systems. Your model needs to split the stream into multiple substreams, group them by account number, run optional transformations on each substream, and finally compute the net transaction value for each substream. Figure 6.6 illustrates this scenario.

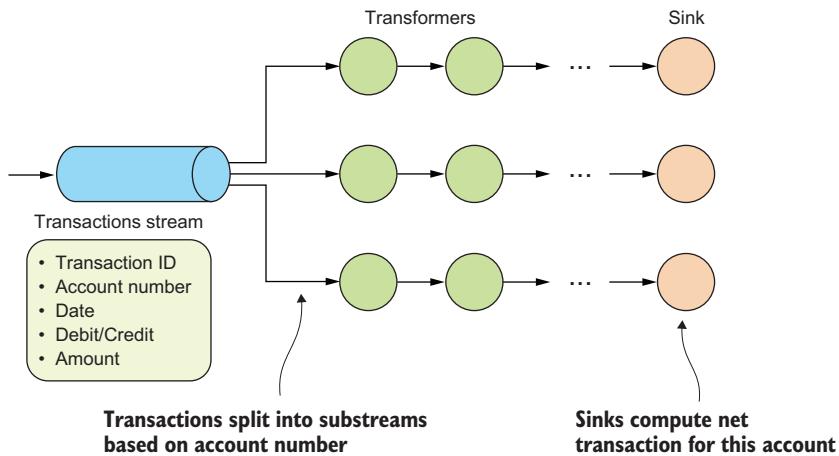


Figure 6.6 The processing pipeline for banking transactions. A stream of transactions comes into the system from upstream, gets split into substreams grouped by account, and after some transformations and validations gets netted into a sink.

You'll use Akka Streams as the implementation framework. This is an example of a framework that's standardized based on the specifications published by <http://reactive-streams.org>. Multiple implementations are based on the same specification, and the ultimate plan is to offer seamless interoperability across all of the reactive stream implementations.

As part of the implementation framework, Akka Streams defines the following basic abstractions for setting up your stream-processing pipeline:

- **Source**—This is where the pipeline starts. A `Source [+Out, +Mat]` takes data from input and has a single output, `Out`, to be written into.
- **Sink**—The pipeline ends here. A `Sink [+In, +Mat]` has a single input, `In`, to be written into.
- **Flow**—This is the basic abstraction where transformation of data takes place. A `Flow[-In, +Out, +Mat]` has one input, `In`, and one output, `Out`. `Mat` refers to the materializer described in the last bullet of this list. So the most basic pipeline consists of one `Source`, a number of `Flows` (which may be 0 in case of a trivial transformation), and one `Sink` (`Source -> Flow* -> Sink`).
- **RunnableGraph**—The entire topology of (`Source -> Flow* -> Sink`) forms a `RunnableGraph` in Akka Streams. The multiple `Flow` structures that can be present between a `Source` and a `Sink` can form junctions that model fan-in or fan-out points of flow. A junction can be of type `Broadcast`, which is a fan-out flow

control. Or it can be of type `Merge`, which models a fan-in. And this whole topology, which is closed and flanked between a `Source` and a `Sink`, is ready to be executed, and is called `RunnableGraph`. You will see `RunnableGraph` in action later in this section.

- **Materializer**—A materializer defines how the transformations are converted into asynchronous processes. For example, `ActorMaterializer` does this transformation through actors.

Back to our use case, here's how to define `Transaction` in our model:

```
sealed trait TransactionType
case object Debit extends TransactionType
case object Credit extends TransactionType
case class Transaction(id: String, accountNo: String,
    debitCredit: TransactionType, amount: Amount, date: Date = today)
```

The next step is to implement the processing pipeline, which you'll do using the abstractions offered by Akka Streams.

STEP 1: SETTING UP THE SOURCE

The processing pipeline begins with setting up the `Source`, which abstracts all the stream-processing steps with a single open output. You start with `Source`, set up the `Flows` by adding transformations to data coming out of the source, and finally materialize the stream into a `Sink`:

```
import akka.actor.ActorSystem
import akka.stream.scaladsl._
import akka.stream._

object TransactionPipeline {
    implicit val as = ActorSystem()
    implicit val ec = as.dispatcher
    val settings = ActorMaterializerSettings(as)
    implicit val mat = ActorMaterializer(settings)

    val transactions: Source[Transaction, akka.NotUsed] =
        Source.fromFuture(allTransactions).mapConcat(identity)
}

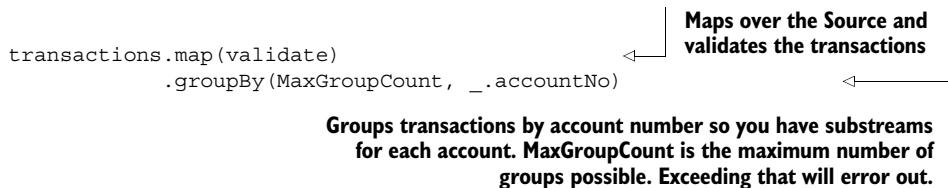
//...
```

All imports and the setup codes should be self-explanatory. The final step sets up the source that starts the pipeline. In this example, `Source` takes input from the `allTransactions` function, which returns `Future`, containing all transactions coming into the system:

```
def allTransactions(implicit ec: ExecutionContext):
    Future[Seq[Transaction]] = Future {
    //...
}
```

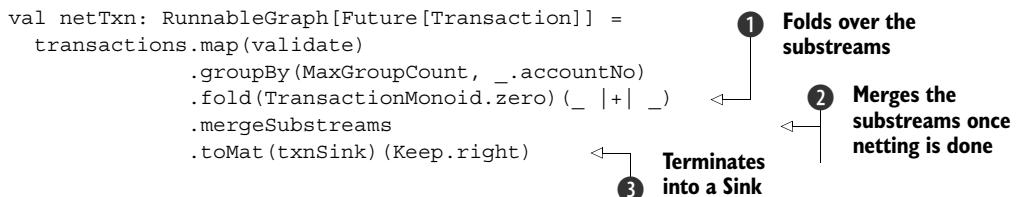
STEP 2: SPLIT THE STREAM INTO SUBSTREAMS

After you've set up the Source, you'll follow the steps in figure 6.5 and split the stream based on account numbers. So you'll have separate streams for each account number, the exact number being determined by the following `groupBy` operation. Before the split, you can plug in transformers for validation or any other domain-specific constraints:



STEP 3: END AT A SINK

As a final step in our small example, you now need to net all transactions per account based on debit/credit flags and terminate our stream to a Sink. The following code does all the steps in sequence and in a completely functional way, using the rich set of combinators that Akka Streams offers. The code is self-explanatory. If you need more details, have a look at the explanation that follows the code.



After you have multiple substreams (one per account), you need to do the following before you come up with a `RunnableGraph` abstraction that you can run. Again note how the stream's API decouples the computation from its execution:

- Run the netting computation over each substream. You do that with a `fold` over the Subflow that `groupBy` returns ①. `fold`⁷ uses the `TransactionMonoid` to accumulate transactions on each substream and sum over the amounts.⁸
- After you're finished with the netting, you merge the substreams ②.
- Finally you materialize into a Sink to get a `RunnableGraph` ③.

Now you have the `RunnableGraph`, which you can run and get your results. In the preceding code, `txnSink` is a Sink, where you materialize your computation. Akka Streams uses a bunch of actors for materializing the computation that you've defined

⁷ The `fold` over the substreams is a sequential operation in this case. If the individual substream operations within the `fold` are expensive, you can parallelize using an explicit `.async` after the `fold`.

⁸ If you need to refresh how a monoid does this, have a look at section 4.1 in chapter 4.

so far. For this simple example, you can define the Sink as one that prints the output on the console:

```
val txnSink: Sink[Transaction, Future[akka.Done]] = Sink.foreach(println)
```

The online code repository has a complete runnable version of this example for you to try.

STEP 4: RUN THE COMPUTATION

You now have the entire RunnableGraph ready to run. Here's a sample run for the entire pipeline that starts reading from the input stream and generating output in the sink:

```
netTxn.run().foreach(println)
```

6.4.2 A graph as a domain pipeline

Let's look at a slightly more complex use case for the stream model, where the sequence of domain behavior invocations can be modeled as a graph. As usual, you'll differentiate between the construction and the execution phases of the graph. As a modeler, you'll focus on the domain behaviors only and how they interact within the pipeline. The underlying framework, Akka Streams, will do the heavy lifting that is behind the execution and management of the graph model.

Figure 6.7 depicts the pipeline that you'll model.

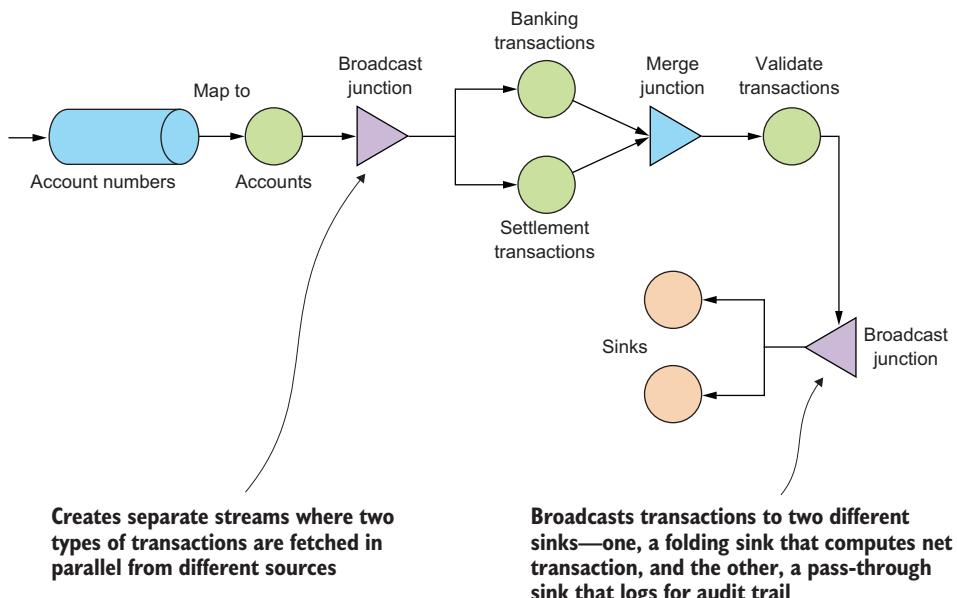


Figure 6.7 The pipeline of activities modeled as a graph in Akka Streams. The accompanying text describes the various stages of the pipeline.

Here are the various steps that the pipeline is supposed to run through during execution:

- 1 It starts with the source getting a stream of account numbers.
- 2 The next stage in the pipeline interacts with the `AccountRepository` and maps the numbers to `Account` objects.
- 3 Then you have a Broadcast junction, which does a fan-out of the incoming stream of accounts into two separate streams. Each of the incoming accounts is emitted simultaneously to both downstream consumers.
- 4 Now you have two separate streams where you apply transformations to fetch transactions for each of the accounts. One gets the banking transactions, and the other gets the settlement transactions that result from cash settlement of security trades. By doing a broadcast, your model can fetch the two streams of transactions in parallel, making optimal use of computing resources.
- 5 After collecting the transactions, you do a merge through a Merge junction. Both streams rejoin to make a single stream of transactions. In general, a merge doesn't preserve the order of inputs in the output. But specialized merge operations such as `MergePreferred` or `FlexiMerge` give you fine-grained control over the ordering.
- 6 You apply another transformation to validate all transactions.
- 7 You do another broadcast. One stream logs all transactions for the audit trail. You model it through a pass-through sink that gets the audit trails without any modification. And another one goes to a sink that does a fold operation to compute the net value of transactions on a per account basis (the folding sink).

Here's the code that constructs the graph for this pipeline by using the graph formation DSL:

```

val graph = RunnableGraph.fromGraph(
  GraphDSL.create(netTxnSink) { implicit b => ns =>
    import GraphDSL.Implicits._

    val accountBroadcast = b.add(Broadcast[Account](2))
    val txnBroadcast = b.add(Broadcast[Transaction](2)) ←
    val merge = b.add(Merge[Transaction](2)) ←

    val accounts = Flow[String].map(queryAccount(_, AccountRepository))
    val bankingTxns = Flow[Account].mapConcat(getBankingTransactions)
    val settlementTxns = Flow[Account].mapConcat(getSettlementTransactions)
    val validation = Flow[Transaction].map(validate) ←

    accountNos ~> accounts
      ~> accountBroadcast
      ~> bankingTxns
      ~> merge
      ~> validation
      ~> txnBroadcast ~> ns
  }
) ←

```

Sets up Merge junction. The argument 2 indicates it's a two-way merge.

Constructs a RunnableGraph, a graph which you can run later with a materializer

Sets up Broadcast junctions. The argument 2 indicates it's a two-way broadcast.

Adds flows

```

accountBroadcast ~> settlementTxns ~> merge
txxBroadcast ~> audit
ClosedShape
})

```



Connects to
form a graph

The preceding code looks intuitive. Have a look at the DSL for the graph construction; it almost replicates the way you'd draw a graph on a whiteboard. After the construction is done, you can run the whole graph by using `graph.run()`. One last piece that you need to look at is the two sinks, which you use for computing net transaction values per account and audit trails. `netTxnSink` nets all transactions per account, and `audit` does the logging. In the current example, `audit` prints on the console, though in a more realistic scenario you'll have more-sophisticated loggers doing this job:

```

val netTxnSink: Sink[Transaction, Future[Map[String, Transaction]]] =
  Sink.fold[Map[String, Transaction], Transaction]
    (Map.empty[String, Transaction]) { (acc, t) =>
      acc |+| Map(t.accountNo -> t)
    }

val audit: Sink[Transaction, Future[Unit]] = Sink.foreach(println)

```

`netTxnSink` is the `Sink` that gets `Transaction` from upstream graph nodes and does a `fold` over them, generating a `Map` that contains per account net transaction values. It uses the fact that `Transaction` is a monoid, and a `Map` whose value is a monoid is itself a monoid. The rest of the implementation should be self-explanatory.

All of this code for the implementations is present in the online code repository for the book.

6.4.3 Back-pressure handling

As mentioned earlier, one of the important aspects that you need to consider in stream-processing models is back-pressure handling. It's a common occurrence that the consumer (`Sink`) can't cope with the throttle that the producer (`Source`) generates. Without proper handling, this can lead to the consumer either dropping data or crashing with an `OutOfMemoryError`.

Akka Streams and other conforming implementations offer strategies to handle back pressure.⁹ While data flows from the `Source` toward the `Sink`, demand flows in the opposite direction. It's the `Sink` that keeps the `Source` updated about the throttle that it can sustain at this time. This is the back-pressure information that continually flows from the consumer (`Sink`) up to the producer (`Source`), so that the producer can moderate the throttle downstream. You can check the details of how the settings for back-pressure handling can be tuned for your model in the documentation.

This chapter has explained the basic working model of a stream-based pipeline. It hasn't yet discussed how this model works across multiple cooperating nodes or an infinite stream of inputs. All inputs that you feed to your sources in the use cases of the

⁹ See Reactive Streams (www.reactive-streams.org).

earlier sections were finite fixed sequences. Also, you haven't yet seen how the back-pressure handling works and how to monitor that it's indeed regulated by the demand that the sink generates for the source. Chapter 7 presents a complete implementation that addresses all these aspects of designing a stream-processing pipeline as part of your domain model. This example combines actors and stream-based models; you'll have actors that feed into the stream-processing pipeline that perform transformations on data based on domain rules and finally terminate in sinks that implement domain behaviors. But first here are the major takeaways from our discussion on the stream model so far:

- *Big and infinite data*—In the age of big data, services that you design often have to consume infinite, or at least large, streams of data. You can't afford to store all data and then process. The stream model is a viable option for this use case.
- *Asynchronous and nonblocking*—Like the future-based computations, streams offer APIs that are asynchronous and nonblocking. This ensures that your model is responsive to the user and that computing resources are utilized optimally.
- *Elastic APIs*—The stream model is based on abstractions such as Source and Sink that are location transparent. You configure your services and point them to the Source of your pipeline. This makes your computation scale.
- *Back-pressure handling*—In the stream model, data flows from a Source to a Sink, while the demand flows in the opposite direction. The consumer (the Sink) signals the demand to the producer (the Source), and based on this demand the producer will regulate the throttle. The Sink continually keeps the Source updated about its demand so that the Source can control the throttle downstream. This handles back pressure and prevents the Sink from being overwhelmed with data volume.

In the next section, you'll take a brief look at the actor model, which is the last concurrency model that you'll visit.

6.5 The actor model

Actors offer a model of computation based on message passing between entities. Any entity can send a message to an actor, which receives the message in its mailbox. On receiving the message, the actor can take any of the following actions:

- Process a message from the head of its mailbox
- Create new actors
- Update its state information
- Send messages to other actors

An actor is usually implemented as an extremely lightweight construct. Typically, you can have millions of actors created in a laptop of standard configuration that can process messages in parallel. Actors were first popularized in the Erlang programming language (www.erlang.org), which uses this model almost exclusively as the primary

concurrency primitive. Inspired by Erlang, Akka brings the actor model to the JVM. Though on some finer points they differ in design, Akka actors are almost equivalent to what you get on the Erlang/OTP platform.

In this section, you'll look at Akka actors mainly from a domain modeling point of view. I won't cover all the capabilities that Akka offers; for that, you can look at Raymond Roestenburg's *Akka in Action* (Manning, 2015). This section highlights the capabilities of the actor model that are useful for designing domain models and points out weaknesses that you may want to avoid. Let's start with some of the primary features that Akka actors offer:

- *Asynchronous message based*—Actors in Akka are asynchronous, message-driven, and nonblocking, which straightaway fits the definition of reactive computation. Messages that you send to actors are usually typed immutable entities that get added to the actor's mailbox. You need to define a receive loop that's capable of handling the types of messages that the actor can potentially process.
- *Single thread of execution*—Actors process one message at a time; it's a single thread of control. You don't need to implement any concurrency within the message loop of an actor. You can have a mutable state within an actor, and that's perfectly fine. Protecting mutable state is one of the frequent use cases of the actor model.
- *Supervision*—Actors in Akka implement supervisor hierarchies. You can define parent-child relationships between actors, in which parents are responsible for management of failing children. When an actor fails with an exception, parents decide whether to restart the child, depending on the supervision strategy that you've declared for your actor system. The primary advantage of this approach is that it makes failure handling a responsibility of the actor infrastructure rather than any specific actor. This is also an idea that Akka has borrowed from the Erlang/OTP platform.
- *Location transparency*—When you send messages to Akka actors, you don't need to know whether the receiving actor is collocated with the sender. You send the message by using the same syntax, and the message is delivered to the actor even though it may reside on a different network node. This is true decoupling between the sender and the receiver.
- *Finite state machines*—You can change the behavior of an Akka actor dynamically by using the become method. This is extremely useful for modeling finite state machines by using the actor model of computation.

6.5.1 **Domain models and actors**

You want to make your domain models reactive, and you've already seen a few options to do so. Actors offer yet another model of concurrency and asynchrony that can lead to scalable and reactive domain models. But this model is a bit different in philosophy and architecture from all other models discussed earlier. This stems from the fact that

actors were primarily conceived as a means to design systems that need to be massively concurrent and scalable. And the fact that they were first popularized in a dynamically typed programming language leads us to believe that actors were never thought of as an architectural construct amenable to easy algebraic reasoning of your model. Do you view this as a drawback when you implement actors in a statically typed programming language? You'll find answers to this question and more as you dig deep into the applicability of the actor model in designing reactive domain models.

ACTORS AND TYPE SAFETY

Akka actors aren't type-safe in their behavior. If you look at the contract for `receive`, which is the primary method that defines the behavior of the actor, it's defined as `PartialFunction[Any, Unit]`, which means that you don't have any static guarantee of what an actor might end up doing. And there are a couple of reasons for this design:

- *FSM*—Akka actors can model a finite-state machine (FSM) by using the `become` method, which implies that the actor should be able to transform its behavior depending on the current state and the message that it receives. This necessitates an intrinsically dynamic behavioral contract. You have more power at the expense of type safety. Keep this in mind when you design your APIs, and try to make them reactive by using Akka actors. This isn't universally evil; there are valid use cases for using this additional power to implement state machines in a succinct, domain-specific language.
- *Distribution*—Akka actors can be distributed across nodes transparently. This requires the ability for actors to be dynamically serializable and hence becomes difficult to impose any static typing guarantees on them.

In contrast, Scalaz also has an actor implementation, which offers more type safety than Akka actors.¹⁰ Actor [A] can receive messages only of type A, where we typically define A to be a sealed trait with multiple subtypes. The compromise that it makes is that a Scalaz actor can neither be used to model an FSM, nor be transparently distributed across nodes.

ACTORS AND COMPOSABILITY

A behavior of type `PartialFunction[Any, Unit]` is all about side effects.¹¹ It's no wonder that Akka actors don't compose because of a lack of referential transparency. Hence you can't reason about Akka actors in your model. Where you can manage with referentially transparent composable APIs, go for that—don't use actors for such use cases.

ACTORS AND REACTIVE API DESIGN

Lack of type safety and composability make actors a not-so-good choice for designing *end-user APIs*, especially when you're using a statically typed programming language

¹⁰ For more information on Scalaz actors, see <https://github.com/scalaz/scalaz>.

¹¹ `PartialFunction[Any, Unit]` means it can take an argument of type `Any` and produce a `Unit`. You can't compose such functions because a type `Unit` doesn't permit composition.

such as Scala. But for some specific use cases, you can have actor-based *implementation* of some end-user APIs. This is especially true when you need to enforce mutual exclusion on a mutable state, and it works like a charm because every actor follows a single-threaded model of execution. This is one of the most recommended use cases for using actors.

Consider the following example, where you'd need to track the maximum valued debit transaction from an asynchronous stream of messages. You can model this tracker as an actor with the maximum valued transaction seen so far being kept as a mutable state. With every message that you receive, you do the checks and change the maximum value if required:

```
class MaxTransactionTracker extends Actor {
    var max: Transaction = _

    def receive = {
        case t @ Transaction(_, _, transType, amount, _) =>
            if (transType == Debit && amount > max.amount) max = t
        case MaxTransactionSoFar => { sender ! max }
    }
}
```

You can now define an end-user API, where the implementation of the API is based on the preceding actor. The actor ensures that your mutable state (the maximum valued transaction) is protected through the single thread of execution and is published only externally through a message to the sender.

In some use cases, you'll find a strong parallel between implementing some behaviors through actors and through futures. Consider the following example of an actor that fetches the portfolio of client accounts for specific instruments:

```
sealed trait Instrument
case class Equity(..) extends Instrument
case class Currency(..) extends Instrument

class PortfolioReporter extends Actor {
    def receive = {
        case GetPortfolio(acc, ins) => //..
        //..
    }
}
```

Now you want to fetch the equity and currency portfolio for a specific client and report them together as a consolidated portfolio. Here's our first attempt (spoiler: a failing one), that uses the `ask` method of the actor that returns a `Future`:

```
val eq = Equity(..)
val ccy = Currency(..)
val pEquity = prActor ? GetPortfolio(acc, eq)
val pCurr = prActor ? GetPortfolio(acc, ccy)
```

```

for {
  e <- pEquity
  c <- pCurr
} yield merge(e.asInstanceOf[Portfolio], c.asInstanceOf[Portfolio])

```

With the knowledge of the actor model that you've seen so far, is this an acceptable model of a reactive API? Actors work in a single thread of execution and though `pEquity` and `pCurr` are `Futures`, the two messages are processed sequentially. You're hitting the same actor here, so you don't have any concurrency in the execution. The solution, as often with the actor model of execution, is to create separate actors to get the desired level of parallelism in execution.

```

val prActor1 = system.actorOf(Props[PortfolioReporter])
val prActor2 = system.actorOf(Props[PortfolioReporter])12
//.. same as earlier

```

Now consider the same use case implemented by using a `Future`-based API without the actor machinery:

```

trait PortfolioReporter {
  def getPortfolio(a: Account, i: Instrument): Future[Portfolio]
}

val pr: PortfolioReporter = //..

val eq = Equity(..)
val ccy = Currency(..)
val prEquity = pr.getPortfolio(acc, eq)
val prCurr = pr.getPortfolio(acc, ccy)

for {
  e <- prEquity
  c <- prCurr
} yield merge(e, c)

```

Isn't this much simpler? You don't have to define actors and the associated boilerplates that come along with them. The API is cleaner and much more composable. So the general recommendation is to prefer `Futures` to actors. `Futures` in Scala are composable and provide lots of combinators for doing declarative programming. In summary, `Futures` compose; actors don't.

ACTORS AND BOUNDED CONTEXT

Earlier in this chapter, you learned how to use asynchronous messaging for communication between bounded contexts. If you're using an actor framework like Akka, you can use the power of actors as well to do this. The actor infrastructure comes with a mailbox and messaging, so you don't have to manage your own queues separately. For

¹² Keeping it simple here, but realistically you can use `RoundRobinPool(2).props(Props[PortfolioReporter])` as a more idiomatic implementation. Refer to Akka documentation for details.

the protocol that you decide for communication, you can design immutable messages and pass them along using actors. Akka also offers location transparency through remote actors and redundancy through clustering. Both of these will help you design a reliable messaging system between bounded contexts.

ACTORS AND RESILIENT MODELS

A full-blown actor model is a coarse-grained abstraction. Actors come with a certain payload, especially if you have features such as supervision, remoting, distribution, and clustering as part of the same framework. Be aware of these consequences before you choose actors as the primary primitive for handling concurrency in your model.

But these additional features that a toolkit such as Akka provides form the cornerstone of some important aspects of your model architecture. When you have a model of nontrivial complexity, you'd like it to be resilient in the face of failures. As discussed in chapter 1, you wouldn't want the user experience to degrade even if some of the components of your application fail. And you can achieve this only if you implement the *Let It Crash* philosophy. Failures will happen within your application. You can address some of these explicitly within your code. But failures in hardware, firmware, or network layers may occur that are out of your application's control. The only way to address failures is by treating them as a separate concern in your architecture and having dedicated components that handle them.

Following the principles of Erlang/OTP, Akka allows supervision to be implemented as a separate architectural construct within your model. You can define supervisor hierarchies within your actors that are dedicated for handling any exception that may occur within the child actors. Depending on the type of exception, you can specify actions such as resume, abort, or restart for the child actors. So if an actor S is declared as a supervisor (parent) for a set of actors [c1, c2, c3], any exception that occurs in the processing of any of the child actors gets propagated to S. S can take actions to address the failures depending on the type of failure. The beauty of this approach is that the behavior of c1, c2, or c3 isn't littered with code that handles exceptions.

ACTORS AND FUNCTIONAL DOMAIN MODELS

From what you've seen so far, actors don't compose, are all about side effects, and have a lot of nondeterminism in the way the model works. Do actors make a good companion as a concurrency primitive when you're designing functional domain models?

As mentioned earlier in this chapter, actors can play an important role, even within a functional domain model, in the following three areas:

- *Protect shared mutable state*—The first one is as a container for shared mutable state, where an actor can guarantee mutual exclusion through its single thread of execution. Now this is something you have deep inside your implementation. The bounding API can be functional and doesn't expose any of the mutable

state that the actor manipulates. This is a pattern of guarded mutation at the implementation level that doesn't impact the purity of the APIs.

- *Resilience*—Use actors as coarse abstractions for modeling architectural constructs such as resilience and redundancy. At the topmost level, have supervisors that protect other top-level service actors. These service actors are mere dispatchers that delegate all domain behaviors to pure functional implementations under the covers. So your functional domain model is pure and yet you have a façade of actors that provide the resilience of your model.
- *Play along with Akka Streams*—Instead of using actors for designing APIs, use the typed model offered by Akka Streams—which is safer and compositional. And then use actors for implementation of those APIs through ActorMaterializer. Then you get the goodness of actors as lightweight and scalable artifacts and yet publish typed APIs for your end user. Section 6.4 discussed a sample use case for streams. You'll see a much bigger use case in chapter 7.

Having said this, many developers are also using the actor model in a more pervasive manner. One such common use in domain-driven design is to model the aggregate root as an actor and implement persistence of domain entities by using Akka Persistence (<http://doc.akka.io/docs/akka/snapshot-scala/persistence.html>). This is one technique that goes well with event sourcing and architectures based on Command Query Responsibility Segregation (CQRS) (discussed in chapter 8). This approach forces you to model aggregate roots as actors as opposed to simple algebraic data types, as you've been encouraged to do so far. As with any other technique, this also comes with its own set of pros and cons. Chapter 8 covers in more detail how this pattern interplays with the spirit of functional modeling and how a simple variation can preserve all the goodness that seemed lost.

6.6 **Summary**

This chapter, which starts part 2 of this book, covered how to make domain models reactive. You learned many options for modeling in Scala. The main takeaways of this chapter are as follows:

- *What is a reactive domain model?* You learned about the characteristics that a reactive model should have and how to realize them in terms of end-user APIs. You explored options available on the JVM and their pros and cons.
- *Futures and promises*—You learned how to design reactive nonblocking APIs by using futures and promises. You looked at a few scenarios from the domain of personal banking and implemented a few patterns for making your model responsive.
- *Asynchronous messaging*—This is one of the most commonly used techniques for communicating between systems that are decoupled in space and time. And multiple bounded contexts in a domain model provide the exact setting in which to use this pattern.

- *The stream model*—Streams provide typed abstractions to model domain behavior pipelines. You learned about use cases from your domain that can be mapped to such implementations. You used Akka Streams for this purpose.
- *The actor model*—This is possibly one of the most commonly used concurrency primitives in Scala brought into its current incarnation by Akka. Although actors can make your domain models scalable, they're not without problems. Actors, which are all about side effects, don't usually get along with the spirit of functional programming. But still they offer some valuable features in protecting shared state and in building fault tolerance within your domain model.



Modeling with reactive streams

This chapter covers

- A reminder about the basic design philosophy of reactive streams
- A complete use case implemented using reactive streams as the backbone
- Ways to combine other techniques such as the actor model with reactive streams
- A detailed discussion on using streams to get the basics of reactive modeling

In the preceding chapter, you learned the basic principles of reactive modeling and techniques for making your models responsive and reactive. In this chapter, you'll take a deep dive into the reactive streams model and an implementation of a use case using Akka Streams. You'll start from the business requirements, map them into the various stages of stream processing, and implement the components step by step. The use case is from our domain of personal banking and involves handling asynchronous boundaries across bounded contexts; this provides a perfect setting for discussing issues such as reactive socket management and back-pressure handling. And the domain behaviors that we discuss involve data transformations,

also suitable for implementation using the streaming model. This chapter covers all aspects of making your model reactive and shows how our implementation addresses each of them.

Figure 7.1 shows a schematic of this chapter's content. It's a guide for you to selectively choose your topics as you sail through the chapter.

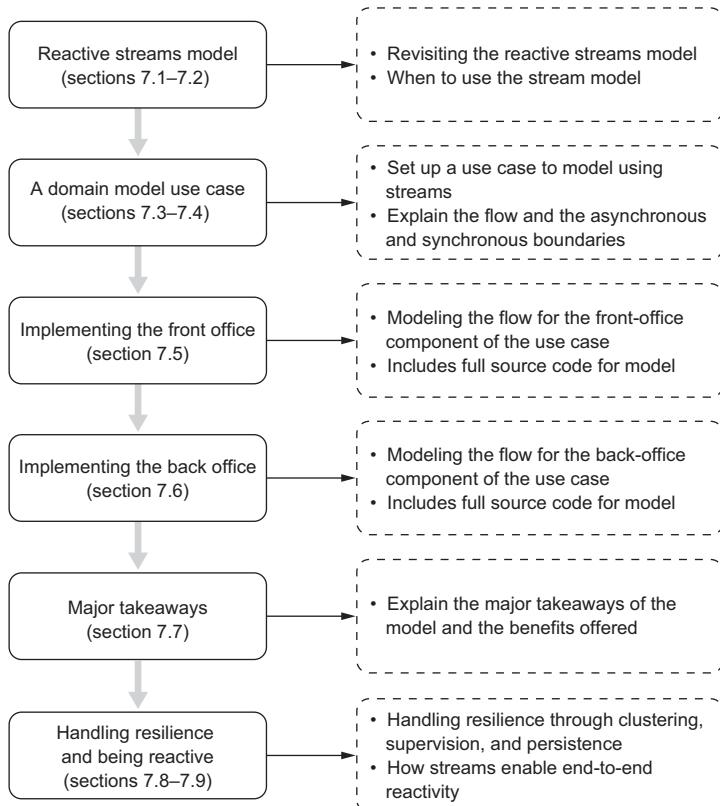


Figure 7.1 The progression of this chapter's sections

At the end of this chapter, you should be in a position to appreciate how to use proper abstractions to make your domain model responsive to users and resilient to failures.

7.1 The reactive streams model

From what you've learned so far, it's not difficult to imagine a domain model as a graph, G , whose *nodes* are the various artifacts such as entities, aggregates, value objects, and services connected by *edges* modeling interactions of domain behaviors. Here we're speaking about a single bounded context with a specific domain and data model, and the interactions modeled by edges can be synchronous or asynchronous.

At a higher level, you can also think of multiple bounded contexts as forming another graph, H, in which each context forms a node, and an edge connecting two contexts models how they interact. The main difference between G and H is that in H, the edges connecting multiple contexts almost exclusively form asynchronous boundaries of the system.¹

In chapter 6, we discussed messaging as one way to handle communication across asynchronous boundaries of a model. You also saw the disadvantages of using the actor model to design APIs based on explicit messaging. Actors offer an untyped model of interaction, which leads to a lack of reasoning of your domain model. But because actors are lightweight, they scale easily; unlike threads, you can have a million actors running on a standard laptop, delivering events that model domain interactions between entities.

The reactive streams model provides the best of both worlds: You get typed APIs that compose as nice, declarative DSLs, which get materialized as implementations of the actor model. You can model synchronous as well as asynchronous boundaries uniformly, handle failures in a principled way, and yet get all the benefits of reasoning with typed compositional APIs. Being reactive, streams handle back pressure that ensures an optimum balance between data flowing from the producer to the consumer, and demand flowing the other way around.

In the next few sections, you'll look at reactive streams in detail and ways to combine them with other models of concurrency. You'll also look at the stream model as a complete solution to building reactive architectures that address the concerns of resiliency, elasticity, and scalability.

7.2 When to use the stream model

Once you realize what a stream model is, the next obvious question is when to use it. After all, streams offer an abstraction that has a certain cost associated with it. Obviously, you'd like to use the cheapest model that best fits your use case. And like any other abstraction, streams don't come free.

You can visualize the dynamic behavior of a domain model as a graph of interactions between processing units. In the vocabulary of stream computing, these processing units are often called *operators*, and the interactions are called *channels*. You can visualize a business use case as streams of data flowing through channels and being transformed through a set of operators. I talk more about such a use case and its stream-based modeling in the next section.

When you have a use case that can be modeled as a series of transformations forming a flow graph, think of reactive streams as a potential abstraction of modeling. The essence of this model is that it makes *flow* a first-class abstraction, thereby decoupling the operators from the concerns of when data arrives or leaves them. Data can be processed asynchronously or concurrently, and your operators are completely oblivious of

¹ For a definition of *asynchronous boundary*, see section 6.3 in chapter 6.

this fact. This makes your model more modular because you've now decoupled the operators from the data flow, and both can evolve and scale independently within your model. As an example, if you look into the architecture of Akka Streams, `Flow` models the stream-processing steps and offers the separate functions `map` and `mapAsync` to characterize the type of data flow (synchronous or asynchronous) between operators. Both `map` and `mapAsync` are different operators that can operate on a `Flow`, thereby making the overall architecture more modular and decoupling the *what* of a `Flow` from *how* it can be manipulated.

7.3 **The domain use case**

Let's set up a use case from our domain of personal banking that we'll model using reactive streams as the main implementation technique. You'll consider the interaction between the front office of the bank that interacts with various external systems and the back office that handles all consolidations and produces the system of records. Here's the workflow that we'll model:

- *Aggregate transactions*—The front office gets a consolidated transaction list after aggregating from various external systems. You'll assume that this aggregation is present with the front office as a comma-delimited file. It could be anywhere (in a message queue or in another persistent store). But for simplicity of implementation, you'll assume a text file.
- *Send to back office*—The front office needs to send the transaction data to the back office for further processing and netting.
- *Create domain-model elements*—The back office receives the transaction information from the file and creates domain objects that can interact with the rest of the model through appropriate domain behaviors.
- *Net transactions*—The back office needs to net transactions per account. A transaction can be a debit transaction or a credit transaction. For a debit transaction, funds will be deducted from the account, and for a credit transaction, they'll be added.
- *Persist and inform*—Netted transactions need to be persisted into a repository. Because this interaction between the front and back offices continues for a long time, the back office needs to publish the status of all netted transactions periodically. In the real world, this publishing may be in the form of dashboards, but here for simplicity you can consider simple prints.

This use case sets up a common model of behavior between interacting systems. You need to handle the asynchronous boundary between the front and the back office. The important point to consider is that this interaction can be quite spiky. During peak business hours, the front office will be loaded with transactions, and the flow of data to the back office will be high. It's important that the back-office system is able to cope with the flow that the interaction demands. And this is one of the primary issues

that reactive streams handle: the back pressure. Let's jump to the model and see how to address this.

7.4 Stream-based domain interaction

So far, we've elided over some of the transformations that need to be done both in the front office and the back office before you can persist the netted transactions. Before detailing them in the upcoming implementation, let's look at a stream-based interaction diagram, which may be more intuitive than a verbose description. Figure 7.2 describes the overall architecture of the model.

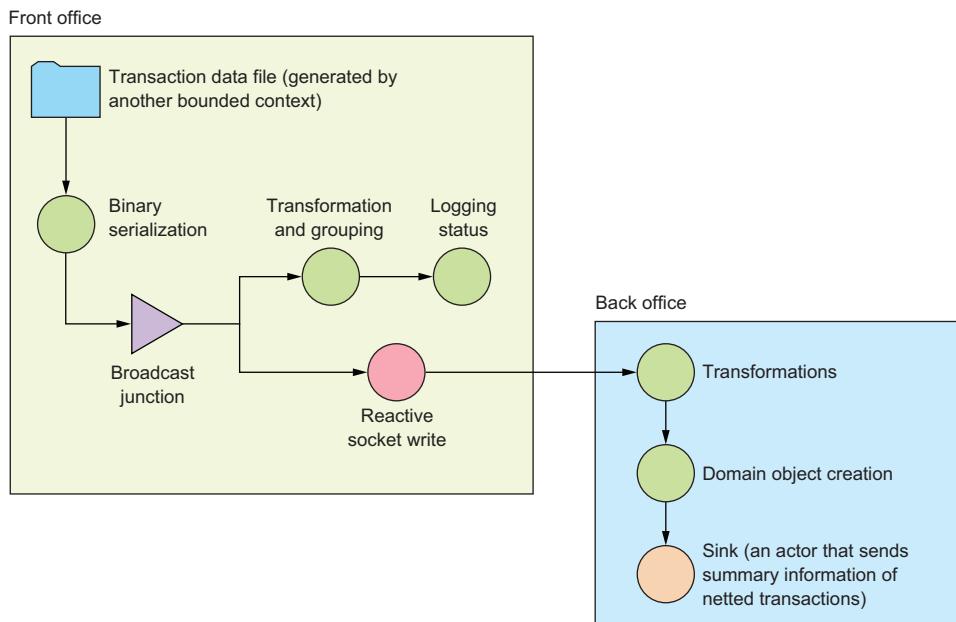


Figure 7.2 Modeling the transaction netting use case using reactive streams. The back-office component has an asynchronous boundary with the front office. The front office receives transaction data in a file that it then transmits to the back office for processing, netting, and persistence.

Let's take a deep dive into the architecture and use Akka Streams to implement the use case (<http://doc.akka.io/docs/akka/2.4.4/scala/stream/index.html>).² After you have the implementation, you can appreciate the value it brings to the table and the ways it improves the programming model by using native actors and explicit message passing. Before you look at the implementation, here are the transformation steps that take place in the processing of front- and back-office systems. The front

² Akka Streams is based on the Reactive Streams specification (www.reactive-streams.org).

and back offices are two separate bounded contexts with strict asynchronous boundaries between them.

- *Serialize to binary*—All transaction data needs to be transmitted to the back-office system. To make this transformation efficient, you serialize into byte streams. This is the first transformation that takes place on the input data. For simplicity, you'll assume that the transaction data has been validated before consolidation into the comma-delimited file.
- *Group and log*—The front-office process starts logging all data that gets transmitted. For optimization, you'll do grouped logging in batches.
- *Transmit*—The front-office process needs to write to a reactive socket for transmission to the back office. A *reactive socket* has a built-in capability to handle back pressure. Note that the receiving side (back office) also binds to the same socket to read the data.
- *Parse and split*—The back-office system receives the binary data and transforms it back to text. It then splits the data based on the comma delimiter.
- *Create transactions*—The next step of transformation takes each individual record and converts them to domain-model objects (`Transaction`).
- *Send to the summarizer for netting*—The summarizer is an `ActorSubscriber`, which is an abstraction in Akka Streams that converts an `Actor` into a stream subscriber with full control of back pressure. It receives all messages (transaction data, in this case) and handles them for processing without ever being overwhelmed by the throughput of the sender.
- *Periodically log the netted transaction*—This is another task that the summarizer does; periodically, it logs what transactions have been netted for every account.

In the next few sections, you'll look at the implementation model based on Akka Streams. After you have the basic implementation working, you'll look at how to add resilience and persistence to this model, all through the various umbrella products of Akka.

7.5 Implementation: front office

Let's look at the front-office operations modeled as a sequence of transformations using the stream model. We already discussed the basic constructs of stream processing that Akka Streams offers in chapter 6 (if you need a refresher, see section 6.4.1.) In the context of our front-office operation implementation, you start with a source, which is the comma-delimited file abstracted into a lazy source:

```
val path = ...           | The path of the file
val getLines = () => scala.io.Source.fromFile(path).getLines()    | Converts the file into a lazy abstraction for iteration
val readLines = Source.fromIterator(getLines).filter(isValid)
  .map( l => ByteString(l + "\n"))                                | Converts into a reactive source, which will be one of the operators in your chain of transformations
```

Here you finally get a `Source`, which generates a stream of bytes that you can transmit down to the back-office system. `isValid` is a validation function that's used to validate every record from the comma-delimited file and filter out the invalid entries. Later you'll see how to deal with invalid records by plugging in strategies to handle exceptions in stream processing.

Now that you have the source, you need to broadcast the contents into multiple paths. One needs to write to the socket for sending out to the back office, and the other logs for heartbeat checks. For heartbeats, you log periodically in groups, so here you need to construct a `Flow` that takes care of this transformation. And this `Flow` will terminate in a `Sink`, which you can ignore for the time being. You're now in a position to weave all these sequences in a graph and run it with a reactive socket connection.

```

val logWhenComplete = Sink.onComplete(r =>
  logger.info("Transfer complete: " + r)) ← A Sink that logs the status
                                                of completion of the
                                                transmission process

→ val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val broadcast = b.add(Broadcast[ByteString](2)) ← Broadcast in
                                                two paths

  val heartbeat = Flow[ByteString]
    .groupedWithin(10000, 1.seconds)
    .map(_.map(_.size).foldLeft(0)(_ + _))
    .map(groupSize => logger.info(s"Sent $groupSize bytes")) ← The Flow that
                                                models the Sink
                                                where you log
                                                the heartbeat

  readLines ~> broadcast ~> serverConnection ~> logWhenComplete
  broadcast ~> heartbeat ~> Sink.ignore
  ClosedShape
}

The overall graph of stream computing. Build
the graph using the mutable builder b. } ← The graph DSL to model
                                                the entire stream
                                                processing

```

You have almost everything you need, except the reactive socket to open the connection and write into and the infrastructure to run your stream computation graph. This is how you open the socket and set up the connection:

```

implicit val system = ActorSystem("front_office")
val serverConnection = Tcp().outgoingConnection("localhost", 9982) ← Opens an outgoing connection,
                                                                which is a back-pressure-aware
                                                                socket over Tcp

```

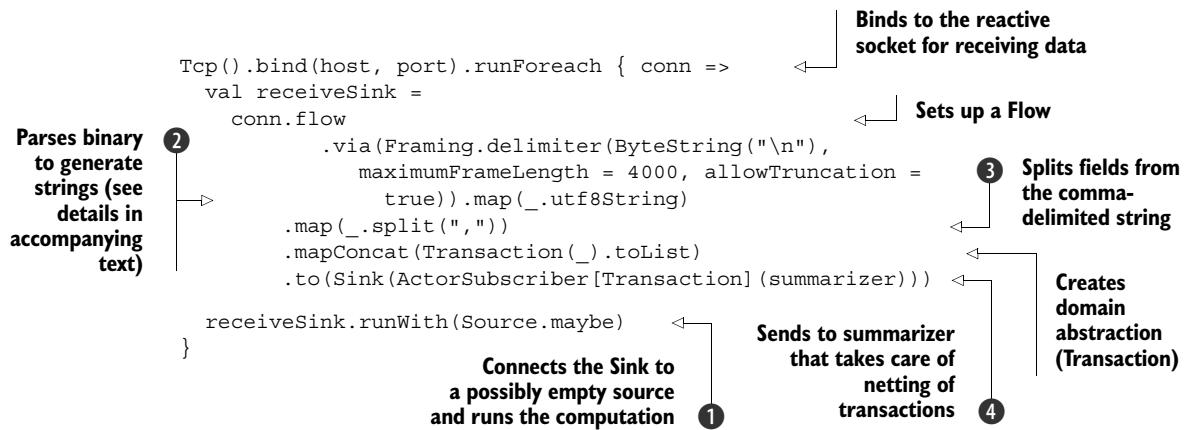
And this is how you can run the graph, using an `ActorMaterializer`. As you saw in chapter 6, an `ActorMaterializer` runs the graph by building actors underneath. Note how the stream-computing framework abstracts the actor model from the user API. As noted in chapter 6, actors can best be used when deployed as an implementation vehicle. The complete version of the runnable code for the front office can be found in the online repository of the book.

7.6 Implementation: back office

This example demonstrates how a combination of various techniques can lead to a solution that's reactive and yet well modularized both within and across bounded contexts. You've already seen the front-office implementation (albeit a simplistic use case) that uses reactive streams as the backbone of implementation. Given a use case, if it fits the requirements of a stream-based implementation (as discussed in section 7.2), you must go for it. Streams, as implemented by Akka Streams, are strongly typed (unlike the actor model) and yet you can get the benefits of scalability that actors offer. Does this mean that you won't use actors explicitly in any of your model elements? As you saw in chapter 6, actors are still useful as an explicit programming model for certain use cases. So, don't get fixed into a specific bias. Use all tools judiciously as the situation demands. In the current use case for a back-office implementation, you'll see how to use a nifty actor to solve a problem that would have been much more circuitous with any other technique.

Section 7.4 listed the tasks (implementation level) that our system is designed for. The last four bullet points cover the items that your back-office implementation will take care of. Let's look at realizing these steps as part of a stream-processing pipeline.

The following listing shows the core steps of transformation that take place after you start receiving data from the front office:



As you saw earlier, APIs that Akka Streams offer are declarative and easily express the designer's intention. Even if you're not aware of the details of implementation, this fragment looks quite intuitive in what it tries to achieve. That's a nice quality to have for an API when you're modeling a domain of nontrivial complexity.

You have an empty `Source` ①, but you don't need one, because you're reading bytes from the socket. The interesting part is the `Flow`, which starts with the parsing of the bytes ②. I won't go into the details of this parsing, because the `Framing` class provides a nice helper that abstracts this well. If you're interested, you can check the documentation and code base for Akka Streams.

After you have the record in string format, you need to split across the comma ❸. This gives you the individual fields for building a Transaction object.

The most interesting stuff happens after you start getting Transaction objects. You need to pass every transaction down to a place that can do the summarization. Note it needs to store the summary records (the netted transactions as Balance) somewhere, which means it needs to manage a state (possibly mutable). It can store the state in memory or in a database. But the important point is that you need to ensure mutual exclusion while manipulating this state. If you've been following the reactive patterns that I've been discussing since chapter 6, this is an ideal use case for an actor. In ❹, summarizer is an actor—but not an ordinary Akka actor. It's a reactive back-pressure-aware subscriber abstraction.³ Let's look at the receive loop of this actor in the following listing.

Listing 7.1 The Summarizer actor is a reactive abstraction

You store netted transactions as a mutable Map. The actor manages single-threaded access to the mutable state.

Our actor is an ActorSubscriber, which means it's a reactive back-pressure-aware actor abstraction.

```
class Summarizer extends Actor with ActorSubscriber with Logging {           ↗
    private val balance = collection.mutable.Map.empty[String, Balance]      ↗

    def receive = {                                ↗
        case OnNext(data: Transaction) =>          ↗
            balance.get(data.accountNo).fold {       ↗
                balance += ((data.accountNo, Balance(data.amount,      ↗
                    data.debitCredit)))               ↗
            } { b =>                                ↗
                balance += ((data.accountNo, b |+| Balance(data.amount,      ↗
                    data.debitCredit)))             ↗
            }
        }
    }
}
```

The receive loop receives Transaction objects.

You net transactions through a monoid. Check the source code in the online repository for how to abstract the logic for handling debit and credit transactions inside a monoid.

So you've now used an actor within your stream pipeline to handle a behavior that is best handled by the actor abstraction. The main takeaway from this implementation is that you should always choose the abstraction that works best for the use case at hand. In this case, you need to model a flow of behaviors in the back office; hence you choose the stream pipeline that gives you declarative graph APIs. Within the whole stream, you need specialized processing capabilities depending on the stage within the pipeline. And you use actors because you need to manage mutable state.

The Summarizer actor manages the state of netted transactions. In the example, you're maintaining a mutable Map that nets the transactions and keeps a Balance per

³ You can look at the Scaladoc for ActorSubscriber in the documentation for Akka Streams.

account. Alternatively, you can update databases to do the same. The last thing we need to address is the requirement to log (maybe in a dashboard) the progress of the netting process. This is something that's totally decoupled from the timeline of the stream-processing sequence and is a completely asynchronous process. Once again, you can use the `Summarizer` actor to do this job with a special message that it receives. And then you can schedule this message dispatch by using some sort of scheduling service. Here's the relevant part of the `Summarizer` actor that does the logging:

```
class Summarizer extends Actor with ActorSubscriber with Logging {
    //... as before
    def receive = {
        case OnNext(data: Transaction) => //... as before
        case LogSummaryBalance => logger.info("Balance so far: " + balance)
    }
}
```

And you can use the back-office processing routine to schedule this message invocation. The following listing shows this and the initial steps of setting up the reactive socket bind along with the full back-office transaction transformation pipeline.

Listing 7.2 Transaction-processing stream pipeline

```
import akka.actor.{ActorSystem, Props}
import akka.stream.ActorMaterializer
import akka.stream.actor.ActorSubscriber
import akka.stream.io.Framing
import akka.stream.scaladsl.{Tcp, Source, Sink}
import akka.util.ByteString

import scala.concurrent.duration._

class TransactionProcessor(host: String, port: Int)
  (implicit val system: ActorSystem) extends Logging {

  def run(): Unit = {
    implicit val mat = ActorMaterializer()

    val summarizer = system.actorOf(Props[Summarizer])

    logger.info(s"Receiver: binding to $host:$port")
    Tcp().bind(host, port).runForeach { conn => ←
      val receiveSink =
        conn.flow
          .via(Framing.delimiter(ByteString("\n")),
              maximumFrameLength = 4000,
              allowTruncation = true)).map(_.utf8String)
          .map(_.split(","))
          .mapConcat(Transaction(_).toList)
          .to(Sink(ActorSubscriber[Transaction](summarizer))) ←
    }
  }
}
```

**Binds to the reactive socket.
Note you use the same host
and port as the front office.**

**The complete
back-office
transformation
pipeline**

```

    receiveSink.runWith(Source.maybe)
}

import system.dispatcher
system.scheduler.schedule(0.seconds, 1.second, summarizer,
  LogSummaryBalance)
}
}

Schedules the logging of summary information using
Akka scheduler. Note it will send a message to the
Summarizer actor every second. The actor will log
the balance from the state that it maintains.

```

7.7 Major takeaways from the stream model

Stream processing is proving to be one of the core techniques for building reactive systems. As we discussed earlier, this model offers all the features of designing non-blocking APIs, which are so critical for making your model reactive. This chapter has focused on implementing use cases that are similar to the ones you use today and yet fit the stream-computing model to a T. You saw two bounded contexts: the front office and the back office, which are spatially and temporally decoupled from each other. Yet they could be nicely integrated by using stream transformation pipelines over a reactive socket. Here's a summary of the major points to take away from this model in general and from the Akka Streams implementation of the Reactive Streams specification in particular:

- *Modeling data that flows like a stream*—For many use cases (including a more general version of this chapter's example), you may get to see the data once. And this model of processing sets up a continuous nonblocking transformation pipeline that works on the data. The details of how this continuity is handled or the flow control is managed is completely abstracted from you. You define your processing pipeline declaratively and publish your demand; the infrastructure takes care of the rest.
- *Declarative*—The APIs are declarative. As you saw in the examples, you use a domain-specific language for stream pipeline implementation. The vocabulary resonates with the ubiquitous language of processing a stream pipeline; for example, you start with a Source, transform through a Flow, and terminate at a Sink.
- *Modular*—Because the APIs are declarative, they allow you to build individual constructs suitable for your domain. A clear separation exists between building the graph and running it. This makes your model components reusable and the model as a whole modular.
- *Concurrent and parallel*—Each processing stage is materialized through actors. Because actors are lightweight abstractions, you can scale a lot with this implementation infrastructure. Moreover, the processing stages are combined effectively so as to deliver the optimal throughput. As an example, successive map stages are pipelined and the communication between the stages is optimized

through internal buffers that they maintain. `mapAsync` is delegated to a `Future` so that expensive operations modeled that way don't block the main thread of execution. When you model a `Broadcast`, stages are executed in parallel again through the actor model, thereby ensuring maximum processing efficiency and improved throughput. But the best part is that the API nicely abstracts all these implementation details, and as a programmer you don't need to bother.

- *Back pressure*—We've talked about this enough by now. The implementation of reactive streams handles back pressure so that the consumer is never throttled with data. Reactive stream implementations such as Akka Streams implement reactive sockets with back-pressure handling built in, so that the topology of your model can adjust to varying loads, depending on the flow of demand and data. This is the essence of elasticity, an important trait for reactive systems covered in chapter 1.

7.8 **Making models resilient**

One of the core principles of a reactive domain model is resiliency—how resilient your model is to failures. This section shows how to make your stream-processing pipeline resilient to failures. Chapter 1 detailed what it means for a model to be resilient. Any failure that occurs shouldn't stall the entire system, except in some major and unlikely scenarios. Failures happen, and it's not possible to safeguard against each type of exception through hardcoded exception-handling code within your domain model.

To address the issue of resiliency, you need centralized management of failures. Something should be able to handle all failures, or at least those that you want to be explicitly handled. You can even specify the action to take when such failures occur. This handler can be hierarchical if you need more-sophisticated error handling; for example, errors can propagate across a supervisor hierarchy and will ultimately be handled at a specific designated level. Erlang (www.erlang.org) implemented this strategy years ago, and Akka has brought the same to the JVM.

This section covers the following aspects that make your model resilient and shows how reactive streams implement each aspect:

- Providing supervised failure management
- Handling failures of nodes or virtual machines
- Making data persistent in the face of failures

We call these the *three legs of resiliency*, the ones that make your complete model runtime resilient to failures. Figure 7.3 provides an illustration. Akka offers all of these through the actor model, and you can use them in your streams-based implementation. The following subsections provide details. For more information on each of these topics within Akka actors, see *Akka in Action* by Raymond Roestenburg (Manning, 2016).

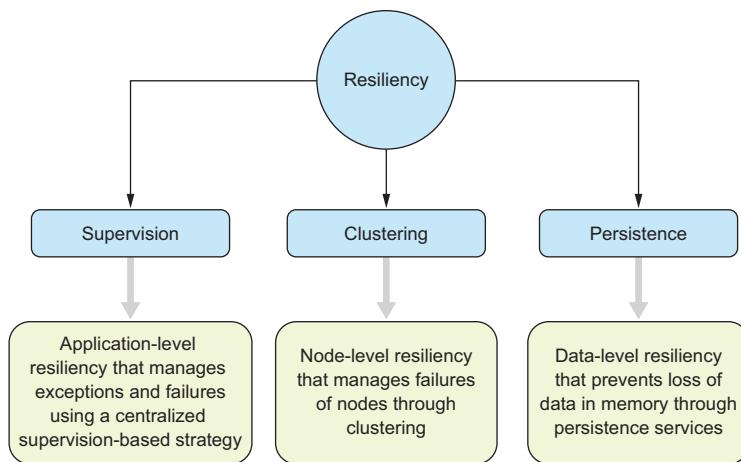


Figure 7.3 The three forms of resiliency. You need to protect against application failures through proper, centralized, supervision-based failure management. You manage node failures through clustering, which Akka offers. And you manage persistence of in-memory data through persistent actors.

7.8.1 Supervision with Akka Streams

You can plug in the supervision strategies that the Akka actor model offers within your stream-based model implementation. The semantics are exactly the same, and the supervisor hierarchies will be in place when you run your application. We briefly discussed the supervision strategies in section 6.5. In the stream-based model presented earlier in this chapter, you can specify the supervision strategy with the materializer:

```

val decider: Supervision.Decider = {
  case _: FormatException => Supervision.Resume
  case _                  => Supervision.Stop
}
implicit val mat = ActorMaterializer(
  ActorMaterializerSettings(system).withSupervisionStrategy(decider))
  
```

In this snippet, you define the exception-handling strategies of your model as a two-pronged approach. When you stream elements (maybe from the front to the back office), if you get any `FormatException` from the records that you read from the Source, then you just drop the element from the stream and continue. That's what `Supervision.Resume` does. Any other exception is treated as fatal, and the processing stops through the strategy `Supervision.Stop`. Besides `Resume` and `Stop`, there's `Supervision.Restart`, which is a bit more fine-grained. Not only is the element dropped from the stream and processing resumes, but the accumulated state of the processing stage is also cleared. The documentation for Akka Streams has all the details.

7.8.2 Clustering for redundancy

A model built using Akka actors can be deployed in a peer-to-peer cluster with no single point of failure or bottleneck. Akka clustering implements an automatic failure detector by using the gossip protocol.⁴ In the current use case, you’re using actors underneath mostly as an implementation technique through the materializers. Hence, when you deploy your stream-based model in a cluster, the resiliency that you get is no different from the resiliency you’d get with an actor-based application model. Because the failure handling is reactive, you get automatic failover to other nodes when a node goes down—an important aspect of a resilient model.

Akka clustering is a complex service, and I don’t describe all the details in this book. For a good overview of the details, see the Akka documentation on clustering.⁵

7.8.3 Persistence of data

Consider a domain model that manages much of its data in memory instead of storing everything in a persistent store. There are valid reasons to do so for some use cases—for example, higher throughput and fewer moving parts to manage. Martin Fowler explains some of these use cases and the variations in model designs on his website.⁶ But while storing data in memory, you need to consider its reliability and persistence too. You can’t afford to lose any data in cases of application restarts, JVM failures, or cluster migrations. You need to use techniques that safeguard against such consequences. And this is the essence of our last point in discussing the resiliency of domain models. In our current use case, the actor `Summarizer` maintains the mutable state of transaction, netting an in-memory Map structure. How do you ensure that the state is persisted even when the application fails?

Akka provides *persistent* actors, which ensure that all internal states of your actor are persisted in permanent storage in an incremental fashion. As soon as changes occur in the state through an event, the event itself is persisted in an append-only store. Note that the whole data is never copied; only the event is kept as part of the log. This process keeps a log of all events that the model has received until that date in chronological order. So now you have the entire history of events that has led up to the current state of the in-memory data structure. You can replay it anytime to reach any state t starting from any state s . It’s the complete audit trail of your whole system. This is a technique that has a more general theory behind it. It’s called *event sourcing* and is one of the popular ways to make reactive models persistent. We cover event sourcing in chapter 8.

⁴ Gossip is a protocol through which the cluster members exchange membership information among themselves to keep all members aware of the topology. You can find more details in the Akka documentation for clustering (<http://doc.akka.io/docs/akka/snapshot/common/cluster.html>).

⁵ Akka Clustering, <http://doc.akka.io/docs/akka/2.3.11/scala/cluster-usage.html>.

⁶ MemoryImage, <http://martinfowler.com/bliki/MemoryImage.html>.

What changes do you need to make in the current example to make the actor persistent? Here's the version of the Summarizer modeled as a `PersistentActor`.

Listing 7.3 The Summarizer as a PersistentActor

```
import akka.persistence.PersistentActor
import akka.stream.actor.ActorSubscriberMessage.OnNext
import akka.stream.actor.{ActorSubscriber, MaxInFlightRequestStrategy}

import scala.collection.mutable.{ Map => MMap }
import scalaz._           ← Summarizer extends PersistentActor
import Scalaz._           ← that comes from Akka Persistence
                           ← and helps persist stateful actors.

class Summarizer extends PersistentActor
  with ActorSubscriber with Logging {
  private val balance = MMap.empty[String, Balance] ← Needs to specify a unique ID
                                                       ← (unique across system). The state
                                                       ← is persisted, indexed by this ID so
                                                       ← that it can be replayed easily by
                                                       ← fetching from the underlying store.

  override def persistenceId = "transaction-netter" ←

  def receiveCommand = { ←
    case OnNext(data: Transaction) => persistAsync(data) { _ =>
      updateBalance(data)
    }
    case LogSummaryBalance => logger.info("Balance so far: " + balance)
  }

  def receiveRecover = { ←
    case d: Transaction => updateBalance(d)
  }

  def updateBalance(d: Transaction) = balance.get(d.accountNo).fold { ←
    balance += ((d.accountNo, Balance(d.amount, d.debitCredit)))
  } { b => ←
    balance += ((d.accountNo, b |+| Balance(d.amount, d.debitCredit)))
  }
}
```

You need to define receive-Command instead of receive, which PersistentActor publishes.

Besides defining `Summarizer` as a `PersistentActor`, you need to specify the details of the underlying storage in the configuration file, where Akka will store the state of the actor. Several journal plugins are available, the details of which are described in <http://doc.akka.io/docs/akka/2.4.4/scala/persistence.html>.

Making an actor persistent enables you to safeguard against data loss, data that you've stored as the state of the actor. This makes your model resilient against application restarts, system failures, and cluster migrations. This also makes your system more scalable, because you've removed one important bottleneck from your architecture: mutation. Instead of updating the persistent store, you're only appending the event that updates the internal state of your model. Appends don't need locking, so they scale better than updates. Also your model becomes much more traceable, because you can now replay events across any timeline and revert back to any snapshot from anytime in the past. Your architecture has audit trails built in. You'll

see more consequences of this architectural pattern in chapter 8, when we talk about event sourcing and related techniques.

7.9 Stream-based domain models and the reactive principles

We've discussed all aspects of modeling your domain by using reactive streams as the backbone. It's now time to look back and see whether this architectural paradigm satisfies all the principles of reactive modeling that we started discussing in chapter 1 and that also are specified in The Reactive Manifesto (www.reactivemanifesto.org). For convenience, let's replicate the 3+1 view of the reactive domain model illustrated previously in chapter 1. Here it is in figure 7.4, annotated to show how reactive streams implement each of the three traits.

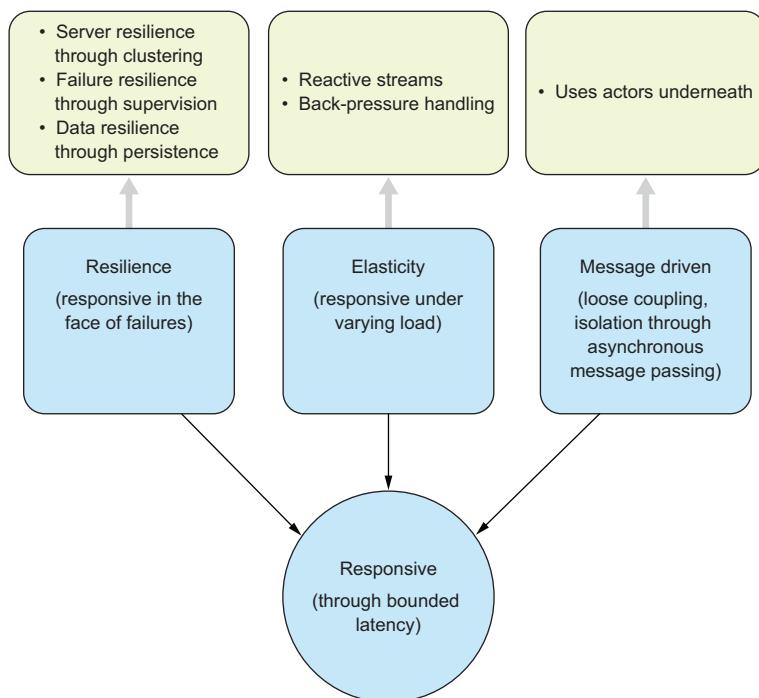


Figure 7.4 Reactive streams implement all the traits of reactive domain models.

Figure 7.4 succinctly explains the various traits of reactive domain models. To summarize this in text, here's how you get an end-to-end reactive model with a streams-based implementation; the three traits that give you a responsive model are all satisfied as follows:

- *Resilience*—With stream-based implementation, you saw three forms of resilience in your domain model. You implemented *resilience from failures* by having centralized failure management using actor-based supervision techniques, which

are available within the reactive streams as well. You got resilience from application failures through *clustering*. Finally, Akka *Persistence* gave you resilience from data loss.

- *Elasticity*—This trait enables you to scale your model in the presence of varying load. A reactive stream implementation with back-pressure handling gives you exactly this; you can control the demand and data flow as per application needs.
- *Message driven*—This is obvious, as you use actors as the implementation underneath.

7.10 Summary

This chapter was a comprehensive introduction to using reactive streams for domain modeling. We started with a business use case and saw a complete implementation of the producer as well as the consumer side. The main takeaways of this chapter are as follows:

- *Using reactive streams for domain modeling*—We discussed the rationale for using stream-based models and how to implement the principles of reactive architecture by using a stream-based backbone.
- *Handling asynchronous boundaries*—Reactive streams make a useful implementation for handling asynchronous boundaries whereby the contexts are decoupled in space and time. Our implementation had this exact use case, as you set up two bounded contexts between the front and the back office.
- *Better than native actors as a programming model*—Akka Streams, an implementation of reactive streams, offers a statically typed programming model where you can reason about the correctness of your architecture. This is better than using actors, which are nondeterministic in behavior. But that doesn't mean you can't use actors as the implementation. Akka Streams uses actors as the underlying implementation of materialized stages of the stream pipeline.
- *Managing resiliency*—You saw how the streams model handles all forms of resiliency: clustering for redundancy, supervision for centralized failure management, and persistence to prevent in-memory data loss.
- *Handling back pressure*—Akka Streams help you control back pressure between the consumer and the producer, thereby preventing the consumer from being throttled by a fast producer.

Reactive persistence and event sourcing



This chapter covers

- How to persist your domain models in a database
- The two primary models of persistence: the CRUD model and the event-sourced model
- The pros and cons of both models and how to select one for your domain model architecture
- An almost complete implementation of an event-sourced domain model
- How to implement a CRUD-based model functionally using a functional-to-relational framework

This chapter presents a different aspect of domain modeling: how to persist your domain model in an underlying database so that the storage is reliable, replayable, and queryable. You've likely heard about storage being reliable, thereby preventing data loss, and queryable, offering APIs so that you can use algebra (such as relational algebra) to query data from your database. This chapter focuses on two more aspects of persistence: traceability and replayability. In these cases, the data store

also serves as an audit log and keeps a complete trace of all actions that have taken place on your data model. Just imagine the business value of such a storage mechanism that offers built-in traceability and auditability of your domain and data model!

Figure 8.1 shows a schematic of the chapter content. This guide can help you selectively choose your topics as you sail through the chapter.

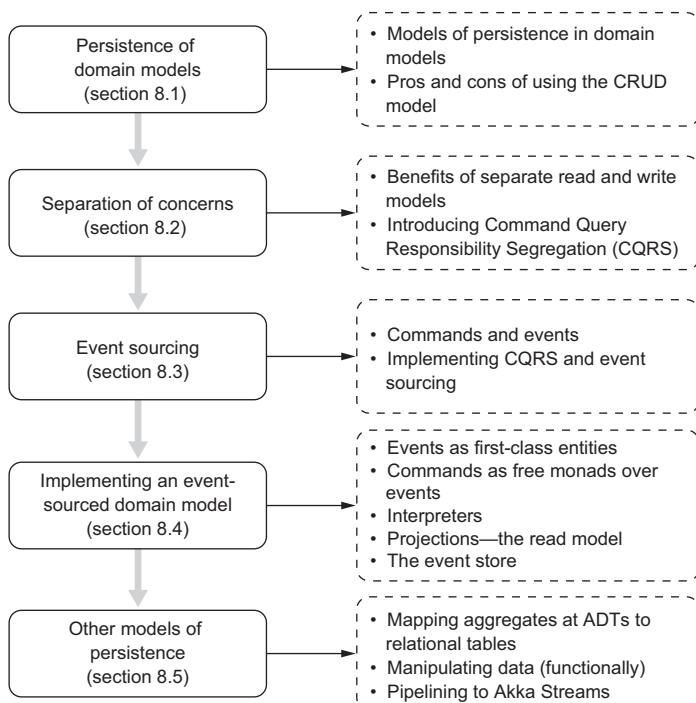


Figure 8.1 The progression of this chapter's sections

At the end of the chapter, you should be in a position to appreciate how to use proper abstractions to make your domain model responsive to users and resilient to failures.

8.1 Persistence of domain models

Chapter 3 presented the three main stages in the lifecycle of a domain object. Any domain object gets *created* from its components, participates in the various roles that it's supposed to *collaborate* in, and finally gets *saved* into a persistent store. In all discussions so far, we've abstracted the concern of persistence in the form of a repository, often with APIs that handle creating, querying, updating, and deleting domain elements. Here's the form of the API that we discussed so far:

```
trait AccountService[Account, Amount, Balance] {
  def open(no: String, name: String, rate: Option[BigDecimal],
          openingDate: Option[Date], accountType: AccountType)
```

```
: AccountRepository => NonEmptyList[String] \ / Account
//...
}
```

This API makes it explicit that you inject a repository as an external artifact in order for open to operate correctly. Now is the time to ask what form of AccountRepository you should use in your model and deploy in production.

In most usage patterns, the common form that a repository takes is that of a relational database management system. All domain behaviors ultimately map to one or more of Create, Retrieve, Update, and Delete (CRUD) operations. In many applications, this is an acceptable model, and lots of successful deployments use this architecture of data modeling underneath a complex domain model.

The biggest advantage of using an RDBMS-based CRUD model as a repository is familiarity among developers. SQL is one of the most commonly used languages, and we also have a cottage industry of mapping frameworks that manages¹ the impedance mismatch between the OO or functional domain model and the underlying relational model.

But the CRUD style of persistence model has at least a couple of disadvantages. An RDBMS often has a single point of failure and is extremely difficult to scale beyond a certain volume of data, especially in the face of high write loads. Concurrent writes with serializable ACID² transaction semantics don't scale beyond a single node. This calls for not only a different way of thinking of your data model, but also an entirely different paradigm to think of the consistency semantics of your domain model.³ I will discuss how using alternative models of persistence can address the scalability problem.

To understand the other issue that plagues the CRUD model of persistence, especially with an underlying functional domain model, let's consider an example model of a CheckingAccount that we discussed in chapter 3:

```
case class CheckingAccount (no: String, name: String,
  dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
  balance: Balance = Balance()) extends Account
```

CheckingAccount is an algebraic data type that's immutable, and you can never do any in-place mutation on an instance of the class. But you already knew this, right? You've seen the benefits of immutability and the dark corners of shared mutability. That's one of the essences of pure functional modeling that we've been talking about for the last seven chapters of this book. Now let's translate this operation to the CRUD-based persistence model. Figure 8.2 shows the result when you perform a debit operation on the account.

¹ Or claims to manage.

² ACID stands for Atomic, Consistent, Isolated, and Durable.

³ Drop ACID and Think About Data (<http://highscalability.com/drop-acid-and-think-about-data>)

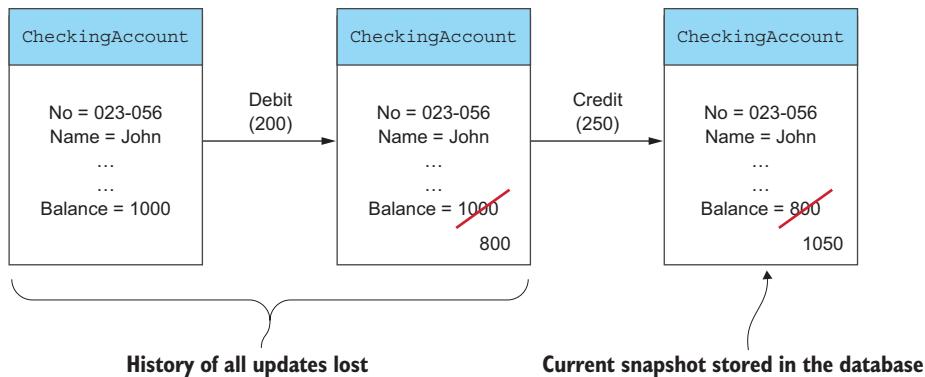


Figure 8.2 Updates in a CRUD-based persistence model using an RDBMS. It stores the current snapshot, losing history of all changes.

You have a table, `CheckingAccount`, that stores all data related to the account. Every time you process an instruction through a domain behavior that impacts the balance of the account, the field `balance` is updated *in place*. Hence after a series of updates, you have only the value that shows the latest balance of the account. You've lost important data! You've lost the entire sequence of actions that has led to the current balance of \$1,050.

This means you won't be able to do the following anymore from your underlying persistent data:

- Query for the history of changes (unless you mine through the audit log that the database stores, which itself is mind-bogglingly complex in nature)
- Roll back your system to sometime in the past for debugging
- Start the system afresh from zero and replay all transactions that have occurred to date, to bring it back to the present

In summary, you've lost the traceability of your system. You have only the current snapshot of how it looks *now*. *Your data model is the shared mutable state*. This chapter presents techniques that allow you to model your repository in such a way that includes time as a separate dimension. The system will have the complete record of changes since inception and in a chronological fashion that offers complete traceability and auditability of the model. Because you've become an expert in functional programming, you can say that the current state of your model is a *fold* over all previous states, with the start state being the initial.

8.2 Separation of concerns

When I discuss models of persistence in this chapter, I frequently draw analogies to aspects of functional domain models and remind us of the lessons we learned there. Let's see whether we can apply some of those techniques at the persistence level as well

and enjoy similar benefits. Separation of concerns is a cross-cutting quality that we consider to be a desirable attribute of any system we design. In my discussion on domain model design, you saw how functional programming separates the *what* from the *how* of your model. You saw how we focused on designing abstractions such as free monads in chapter 5 that separate the definition of your abstraction from the interpretation. When we talk about persistence of domain models, the two aspects that demand a clear separation are the *reads* and the *writes*. In this section, you'll learn how to achieve this separation and get better modularity of your data model.

8.2.1 **The read and write models of persistence**

A user who wants to view a model usually likes to have a business view of it. For example, a user who views an account wants to see all attributes as per the format and guidelines that the business domain mandates. This underlying Account aggregate may consist of multiple entities, possibly normalized for storage. But as a user, I'd like to see a denormalized view based on the *context*. If I'm an account holder in the bank and want to view the balance of my account, I should be able to get to the details that I need for online banking purposes. Another user of the system may be interested in a snapshot purely from an accounting perspective; maybe she's interested only in the attributes of the account that she needs to send to an ERP system downstream for accounting purposes. Here we're speaking of two views of the same underlying aggregate, one for the online banking context and the other for the accounting context. And this is again completely independent of the underlying storage model. Figure 8.3 gives an overview of this separation of the two models.

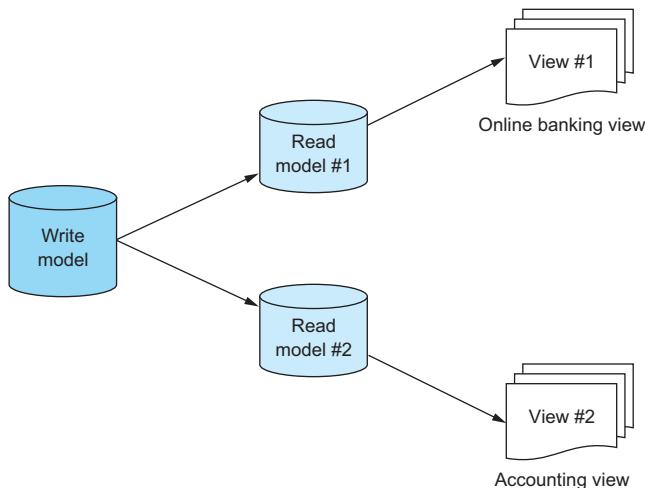


Figure 8.3 Separating the read models from the write model.
A single write model feeds the read models. The feed depends on the bounded context and the information that the specific read model requires.

So you've achieved one level of separation in your underlying data model. All reads will be served from the read models, and all writes will be done through the write

model. From the system-engineering point of view, if you think a bit, you'll see that this separation makes complete sense. For reads, you'd like to have underlying data represented more closely to the queries and reports that they serve. This is the denormalized form that avoids joins in relational queries and just the opposite of what you'd like to do for writes. So by separating the two concerns, you get more appropriate models at the read and write levels.

Reads are easier to scale. In a relational database, you can add read slaves depending on your load and get almost linear scalability. Writes, on the other hand, are a completely different beast altogether. You have all the problems of shared mutable state if you employ a CRUD-based write model. The difference with this domain model is that here the RDBMS manages the state for you. But being managed by someone else doesn't imply that the problems disappear; they just move down one level. In the course of this discussion, I'll try to address this issue and come up with alternate models that don't have to deal with the mutation of persistent data.

The architecture of figure 8.3 has quite a few issues that we haven't yet addressed:

- How will the read models be populated from the write model?
- Who is responsible for generating the read models and fixing the protocols that will generate them?
- What kind of scalability concerns does the preceding architecture address?

We'll address all of these concerns shortly. But first let's take a look at a pattern at the domain-model level that segregates domain behaviors that read data (*queries*) from the ones that update aggregates and application state (which we call *commands*), and that makes good use of the underlying read and write persistence models.

8.2.2 **Command Query Responsibility Segregation**

Let's get back to our domain model of personal banking and look at some of its core behaviors. A few examples are opening an account with the bank, depositing money into my account, withdrawing money from my account, and checking the balance. The most obvious difference in these behaviors from a modeling perspective is that the first three of them involve changing the state of the model, while checking the balance is a read operation. To implement these behaviors, you'll work with the Account aggregate that will be impacted as follows with these operations:

- *Open a new account*—Creates a new aggregate in memory and updates the underlying persistent state
- *Deposit money into an existing account*—Updates balance of an existing account aggregate
- *Withdraw money from an existing account*—Updates balance of an existing account or raises an exception in case of insufficient funds to withdraw
- *Check the balance*—Returns the balance from an account; no application state changes

Of these operations, the ones that change the application state are called *commands*. Commands take part in domain validation, updating in-memory application state and persistence at the database level. In summary, commands impact the write model of our architecture in figure 8.3. The operation of checking the account balance is a read or a *query* that can be served well through the read model of our architecture.

Command Query Responsibility Segregation (CQRS) is a pattern that encourages separate domain model interfaces for dealing with commands and queries.⁴ It provides a separation of reads and writes at the domain model and the persistence model. Figure 8.4 illustrates the basic CQRS architecture.

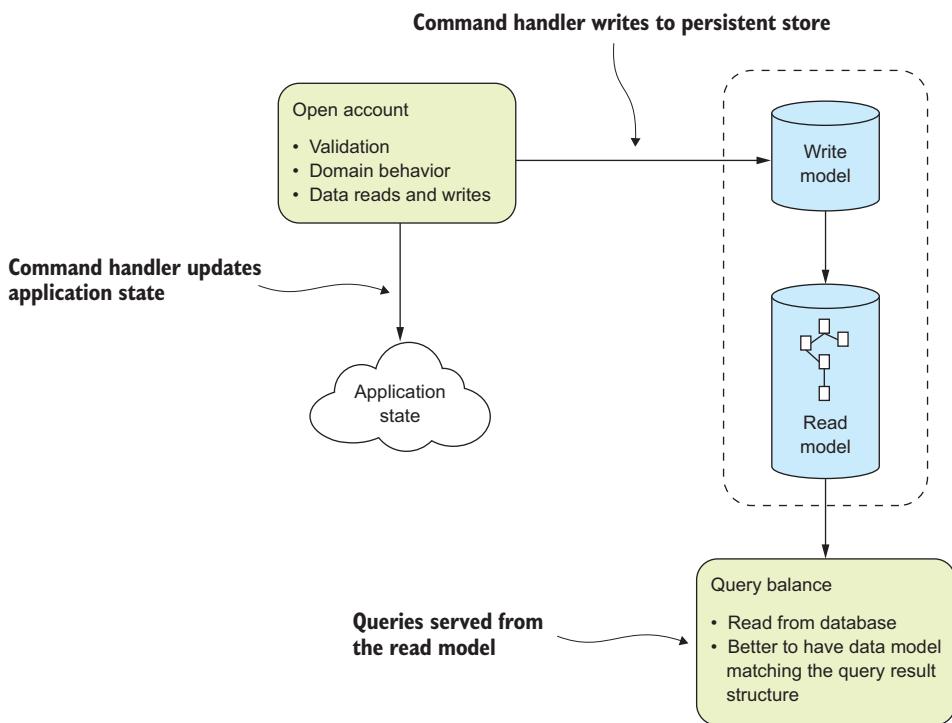


Figure 8.4 Commands update the application state and persist in the write model. The queries, on the other hand, use the read model.

As you can see, the CQRS pattern applied at the domain-model level nicely complements the idea of its dual model at the persistence level. Figure 8.4 is self-explanatory, but here are a few more things that you need to be aware of while applying the CQRS pattern. Strictly speaking, these aren't drawbacks of the pattern, but you need to take

⁴ For more information on the CQRS pattern, see <https://msdn.microsoft.com/en-us/library/dn568103.aspx>.

a careful look and decide whether any of these constraints can be a potential bottleneck in your system:

- It's not mandatory that you have two physically separate databases for read and write models. You can use a normalized relational schema for the write model and a bunch of views for the read model—all part of the same database.
- Depending on your application requirement, you may have multiple read models with a single underlying write model. Just keep in mind that the read models need to serve queries. Hence the schema of a read model needs to be as close to the query views as possible. This need not necessarily be relational; your write model can be relational, whereas the read model can be served from something completely different (for example, a graph database).
- In some cases, the read models need to be explicitly synchronized with the write model. This can be done either using a publish-subscribe mechanism between the two models or through some explicitly implemented periodic jobs. Either way, a window of inconsistency may arise between your write and read models. Your application needs to handle or live with this eventual consistency.

So now you have separate write and read models at the database level and commands and queries at the domain-model level, with clear and unambiguous lines of interaction defined between them. The next section introduces another pattern that nicely complements CQRS and addresses some of the drawbacks of the CRUD model.

8.3 Event sourcing (events as the ground truth)

In the CQRS architecture, the write model faces all online transactions that can potentially involve mutation of shared data. With a typical CRUD-based data-modeling approach, this has all the sufferings and drawbacks that we discussed in section 8.2. Updates (the *U* in CRUD) involve mutation of shared state implemented through locking and pose a serious concern in scalability of the model, especially in the face of high write loads. And as you saw in section 8.2 by in-place updating of data, you lose information of historical changes that take place in your system.

Event sourcing is a pattern that models database writes as streams of events.⁵ Instead of a snapshot of an aggregate being stored in the database as in the CRUD model, you store all changes to the aggregate as an event stream. You have the entire sequence of operations that has transformed the aggregate from inception to the current state. With this information, you can roll back and obtain a snapshot of the aggregate at any time in the past and again come back to the current snapshot. This gives complete traceability of the entire system for purposes such as auditing and debugging. Figure 8.5 describes how to combine CQRS and event sourcing as the foundation of our persistence model.

⁵ For more information on the event-sourcing pattern, see <https://msdn.microsoft.com/en-us/library/dn589792.aspx>.

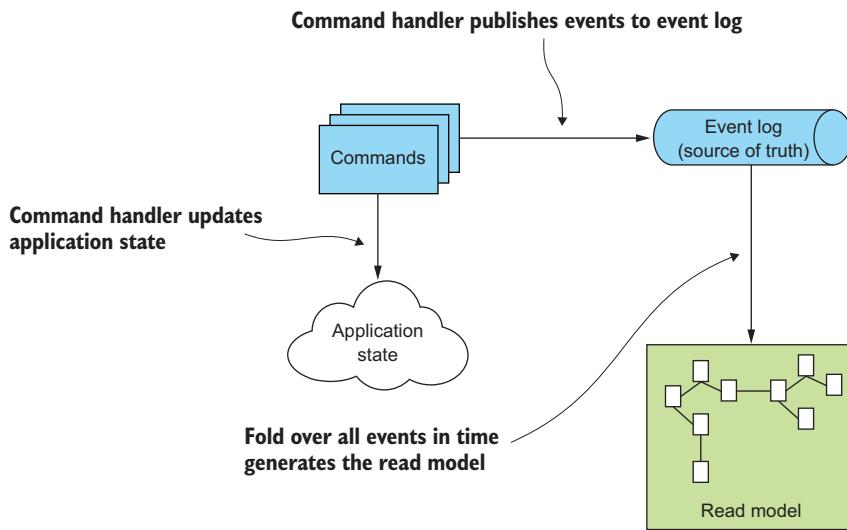


Figure 8.5 Events as the source of truth that get folded over to generate the current snapshot of the system. This generates the read model used by queries.

8.3.1 Commands and events in an event-sourced domain model

As figure 8.5 illustrates, here's how the domain model behaviors interplay with the underlying persistence model, using events as the primary source of truth:

- The user executes a command (for example, opening an account or making a transfer).
- The command needs to either create a new aggregate or update the state of an existing one.
- In either case, it builds an aggregate, does some domain validations, and executes some actions that may also include side effects. If things go well, it generates an event and adds it to the write model. We call this a *domain event*, which we discuss in more detail in the following subsection. Note we don't update anything in the write model; it's essentially an event stream that grows sequentially.
- Adding an entry in the event log can create notifications for interested subscribers. They subscribe to those events and update their read models.

In summary, events are the focal points of interest in an event-sourced model. They persist as the source of truth and get published downstream for interested parties updating read models. Let's discuss in more detail what domain events look like and the exact role that they play in our proposed model.

DOMAIN EVENTS

A *domain event* is a record of domain activity that has happened in the past. It's extremely important to realize that an event is generated only *after* some activity has

already happened in the system. So an event can't be mutated; otherwise, you'd change the history of the system. Here are a few defining characteristics of an event in a domain model:

- *Ubiquitous language*—An event must have a name that's part of the domain vocabulary (just like any other domain-model artifact). For example, to indicate that an account has been opened for a customer, you can name the event `Account-Opened`. Or if you have the proper namespaces for modules, you can have a module named `AccountService` and have an event named `Opened` within it. Also note that the name of an event needs to be in the past tense, because it models something that has already happened.
- *Triggered*—An event may be generated from execution of commands or from processing of other events.
- *Publish-subscribe*—Interested parties can subscribe to specific event streams and update their respective read models.
- *Immutable*—An event being immutable is usually modeled as an algebraic data type. You'll look at the exact implementation when I discuss one of our use cases.
- *Timestamped*—An event is an occurrence at a specific point in time. Any data structure that models an event has to have a timestamp as part of it.

Figure 8.6 illustrates an example use case; specific commands are executed as part of account services, and events are logged into the write model. This simple use case explains the basic concepts of how commands can generate events that get logged into the write model and help subscribers update their read models. I've deliberately

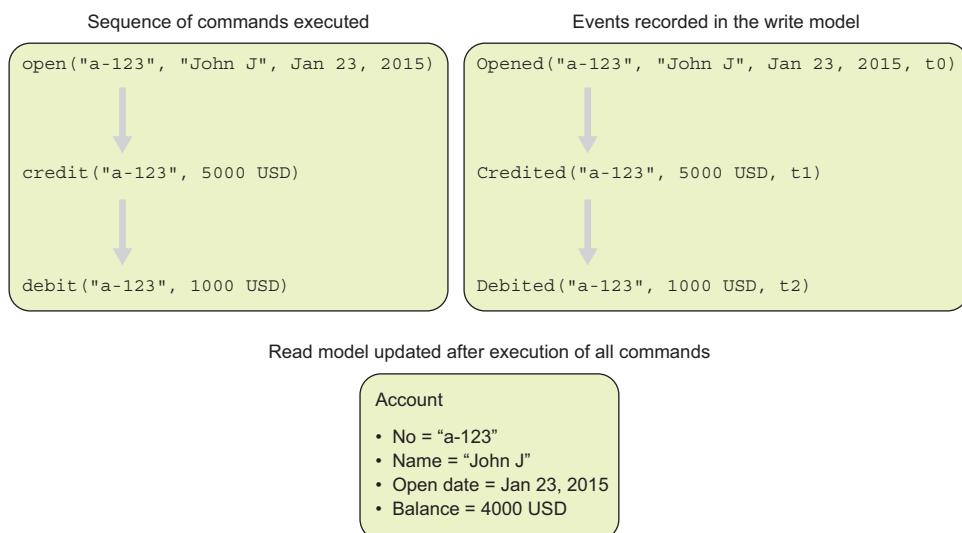


Figure 8.6 Commands generate events that get logged in the event log (write model). Parties can subscribe to the event stream and update read models. Note the naming of domain events and how they belong to the domain vocabulary.

chosen a familiar use case so that you can relate to the examples presented in earlier chapters.

COMMAND HANDLERS

Commands are actions that can loosely be said to be the dual of events. At the implementation level, we talk about command handlers when we abstract the domain behaviors that a command could execute. These include creating or updating aggregates, performing business validations, adding to the write model, and doing any amount of interaction with external systems, such as sending email or pushing messages to the message queue. You know how to abstract side effects functionally, and you'll use the same techniques in implementing command handlers. Command handlers are nothing more than abstractions that need to be evaluated for performing a bunch of actions leading to events being generated and added to the stream (write model). Figure 8.7 describes the flow of actions within a command handler that implements a function to debit an amount from a customer account.

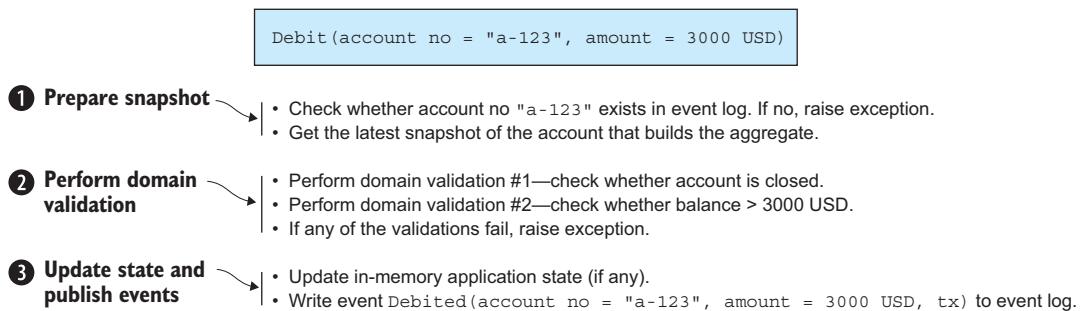


Figure 8.7 A command handler that debits from a customer account. Note the three main stages involved in the flow within the command handler.

8.3.2 *Implementing CQRS and event sourcing*

As mentioned earlier in this chapter, event sourcing and CQRS are often used together as a pattern of persistence in domain models. Figure 8.6 shows a sample flow that's executed when a command is processed and events are generated. But quite a few subtle points make the implementation model not so simple, especially if you want to keep the model functional and referentially transparent. This section presents some of these issues, followed by sample implementations in the upcoming sections.

Just as a recap, here's the flow of commands and events through a CQRS- and ES-powered domain model, with added details:

- You fork a command through a service interface (which can be UI or REST endpoints or any other agent that invokes a domain service).

- If the command needs to work on a new instance of an aggregate (for example, creation of a new account), it checks from the event store that the instance hasn't already been created. Bail out if any such validation fails.
- If the command needs to work with an already existing aggregate (for example, a balance update), it needs to create the instance either from the event store through event replays or get the latest version of it from the snapshot store. Note that the snapshot store is an optional optimization that stores the current snapshot of every aggregate through periodic replays from the event store. But for large domain models it's always used for performance considerations.
- The command processor then does all business validations on the aggregate and the input parameters and goes on processing the command. Note that command processing involves two major steps:
 - Perform domain logic that changes the states of aggregates such as setting the closing date of an account in case of an account close command, and may also involve side effects (for example, sending out email or interacting with third-party services).
 - Generate events that get logged into the event store. In response, subscribers to specific events update their read models.
- If you need to rebuild specific aggregates from scratch, you have to replay events from the event log. This may sound simple, but remember, replaying events requires only re-creating the state and *not* repeating any side effects that the commands may have performed earlier. I'll discuss in the implementation model how to decouple side effects (which only command handlers need to execute) from the state changes (which both command and event handlers need to execute).

BATCHING COMMANDS

We've always valued compositionality; why should we forego it for commands? After all, commands are APIs, and we'd like to make them compositional. You should be able to compose larger commands out of smaller ones, just as we discussed composition of APIs in earlier chapters. Here's an example from our domain. Suppose you have individual commands for debit and credit of accounts. Composing a transfer command by combining the two should be possible:

```
def transfer(from: String, to: String,  
            amount: Amount): Command[Unit] = for {  
    _ <- debit(from, amount)  
    _ <- credit(to, amount)  
} yield ()
```

Remember, you got similar compositionality with our domain service APIs in chapter 5. Do you recollect the technique you used? Yes! Free monads.⁶ They're ready to

⁶ If you need a refresher, feel free to jump back to chapter 5 and read the sections on free monads.

make a comeback in our implementation of event-sourced domain models. You'll define commands as free monads over events so that you can combine commands monadically to build the algebra of our model behavior. You can learn all the details in the next section.

HANDLING SIDE EFFECTS

One extremely important issue to solve if you want to keep your model pure and referentially transparent is how to handle side effects. Execution of command handlers will have side effects, and you'd like to decouple them from the state-changing APIs. Event handlers when replayed need to change states, and this can't induce side effects. You wouldn't want your customers to receive email every time you replay events within your model.

After you use free monads for command composition, the interpreter of the free monad can handle all your side effects. And because commands are the free monads, the side effects are limited to execution of commands only. You have to extract the state-change APIs as separate functions that can be reused separately by the command and event handlers. This is just an overview of the strategy that you'll adopt. The implementation in the next section gives you all the details.

8.4 *Implementing an event-sourced domain model (functionally)*

This section covers some of the implementation aspects of an event-sourced domain model using the domain of personal banking. We use some of the use cases from earlier chapters so that you can relate to the functionality and think in terms of persistence of those model elements using the paradigm of event sourcing. This is by no means a production-ready, event-sourced domain model implementation; we'll simplify wherever we can. The idea is to give you some food for thought for a functional implementation and highlight some of the relevant issues that may arise.

As hinted in the previous section, you'll use free monads to implement event sourcing. Not that this is the only way to implement event sourcing (many other alternative techniques exist as well), even with functional programming. But I choose this approach mainly because it offers a nice separation of the algebra of events from their interpretation. You can keep on building sequences of commands that are pure, and only when you have the final stack of commands can you submit for evaluation through the interpreter of the algebra. This makes the whole model more compositional and restricts side effects only inside the interpreter—just what the doctor for FP ordered.

Here's our general strategy of implementation:

- Define the algebra of events by defining each event as an algebraic data type that can be connected to each other. You saw how to do this in chapter 5, when you implemented a free-monad-based account repository.
- Make a free monad out of the event; this monad is the command. So now you can combine commands by using for-comprehensions and build larger commands.

This is phase 1 of our implementation that builds up our command stack as a pure data type with no denotation.

- In phase 2 of our implementation, you execute actions corresponding to each of the commands. Here you design an interpreter that takes the whole command stack, traverses it, and does a pattern match on the events associated with each command. For every event, you implement the action that it's supposed to execute for processing that command.

Event sourcing using free monads—the algebraic interpretation

Commands are the free monads that build up your data type (which is pure), and then you use the algebra of the events to interpret the data type and execute the commands.

8.4.1 Events as first-class entities

A domain event is a first-class entity in the entire paradigm of event sourcing. The entire model is based around domain events, the model history is stored as event streams, and model (re)generation is done via event replays. It's no surprise that we reflect the same fact in our domain model implementation. The following listing provides the basic Event abstraction and the algebra for the events that you plan to handle in our AccountService implementation.

Listing 8.1 Algebra of events

```
import org.joda.time.DateTime
import cqrsservice._
import common._

trait Event[A] {
    def at: DateTime
}

case class Opened(no: String, name: String,
    openingDate: Option[DateTime],
    at: DateTime = today) extends Event[Account]

case class Closed(no: String, closeDate: Option[DateTime],
    at: DateTime = today) extends Event[Account]

case class Debited(no: String, amount: Amount, at: DateTime = today)
    extends Event[Account]

case class Credited(no: String, amount: Amount, at: DateTime = today)
    extends Event[Account]
```

Base abstraction for an event

The time at which the event was fired

The complete algebra for all events you want to handle as part of the use case. This part is entirely specific to the Account aggregate.

Here, all events are named in the past tense, indicating that they've already occurred. Hence they're immutable, as can be inferred from the algebraic data types defined for

each of them. And finally, the names of the events are from the domain vocabulary, which is perhaps something that you've learned to accept as the norm by now.

Now that you know what each of the events looks like in delivering an account service to the customer, let's step back a bit and think about how these events are generated in the first place. Commands get executed, and as discussed in section 8.3.2, the states of the aggregates change in the course of those actions. And if a command executes successfully, you generate an event. In the same section, you also saw that the state of an aggregate may change in the course of event replays. Let's see how to model the APIs that can be used to affect the state change of an aggregate in both of these situations.

You'll define two functions in a module named `Snapshot`, because the basic purpose of changing the state of an aggregate is to generate its current snapshot. The following listing introduces a few basic common types and defines the module `Snapshot`.

Listing 8.2 The API for state change of an aggregate and generated snapshot

```
import scalaz._  
Import Scalaz._  
  
object Common {  
    type AggregateId = String  
    type Error = String  
}  
  
import Common._  
  
trait Aggregate {  
    def id: AggregateId  
}  
  
trait Snapshot[A <: Aggregate] {  
    def updateState(e: Event[_], initial: Map[String, A]): Map[String, A]  
  
    def snapshot(es: List[Event[_]]): String \/ Map[String, A] =  
        es.reverse.foldLeft(Map.empty[String, A]) { (a, e) =>  
            updateState(e, a).right  
    }  
}
```

The code is annotated with several callout boxes:

- A bracket on the right side groups the `updateState` and `snapshot` definitions, with the annotation: "The function that updates the state of an aggregate from an event. It picks up the aggregate ID from the event, fetches the current state from the rolling snapshot supplied to it, and updates the state based on the event. Note this function is specific to the aggregate and has to be defined separately for each aggregate."
- An annotation pointing to the `Common` object definition: "Some basic type definitions".
- An annotation pointing to the `Aggregate` trait definition: "The base module for an Aggregate. It has to have an ID, which must be unique."
- An annotation pointing to the `snapshot` function: "The generic snapshotting function. Takes a list of events, replays them starting from an empty state, and comes up with a snapshot for all the aggregates contained in the list of events."

In listing 8.2, the function `updateState` is like a `State` monad.⁷ It takes an initial state and an Event and updates the state of the aggregate to the current snapshot. The processing within `updateState` depends on the aggregate and needs to be implemented as part of the aggregate-specific functionality. The function `snapshot` is generic; it's just a fold over the list of events supplied and generates the current snapshot for all the participating aggregates.

⁷ I discussed the `State` monad extensively in section 4.2.3 in chapter 4.

The next obvious step is to give you an idea of how `updateState` works for our use case, as shown in the following listing.⁸

Listing 8.3 The state-changing API for the Account aggregate

```
object AccountSnapshot extends Snapshot[Account] {
    def updateState(e: Event[_], initial: Map[String, Account]) = e match {
        case o @ Opened(no, name, odate, _) =>
            initial + (no -> Account(no, name, odate.get))           ←
        case c @ Closed(no, cdate, _) =>
            initial + (no -> initial(no).copy(dateOfClosing =
                Some(cdate.getOrElse(today))))                                Every event
                                                                updates the
                                                                aggregate state
                                                                within the Map.

        case d @ Debited(no, amount, _) =>
            val a = initial(no)
            initial + (no -> a.copy(balance =
                Balance(a.balance.amount - amount)))
        case r @ Credited(no, amount, _) =>
            val a = initial(no)
            initial + (no -> a.copy(balance =
                Balance(a.balance.amount + amount)))
    }
}
```

You've already accomplished quite a bit of the implementation of our event-sourced domain model. Let's look at the summary of our achievements in this section with respect to our implementation:

- Defined the algebra of events for processing accounts (listing 8.1).
- Defined a state-changing API that can be used to change the state of an aggregate, depending on the event (listing 8.2). You'll see how to use it when you implement command handlers.
- Defined an API for snapshotting that uses the state-changing API in the preceding bullet point (listing 8.3). This gives you an idea of how to use snapshotting to re-create a specific state of an application domain model. For example, you can pick up a list of events between two dates and ask your model to prepare a snapshot. Note you can do this only because you have the entire event stream that the system has ever processed.

In the next section, you'll move on to the more interesting part: command processing.

8.4.2 Commands as free monads over events

Every command has a handler to perform the actions that the command is supposed to do. In a successful execution, an event is published in the event log. So a command

⁸ We use `Account` as an aggregate. For brevity, the `Account` definition isn't included here. The online repository for chapter 8 code provides details. The exact definition of `Account` isn't important for understanding how states change.

executes actions, and you model a `Command` as a free monad over `Event`. Your command-processing loop will be modeled in the following way:

- A command is a pure data type. Because it's a monad, you can compose commands to form larger commands. This composite command that you build is still an abstraction that has no semantics or operation.
- All effects of command processing take place in the interpreter of the free monad when you add semantics to the data type. You'll use the algebra of events to find the appropriate command handler and execute actions and publish events.

If this sounds complicated, wait until you see the implementation. If you understood the explanation of free monads in chapter 5, you'll be able to relate to the same concepts here as well. And along the way, you'll discover the virtues of a purely functional event-sourced domain model implementation. The following listing provides the base module of your `Command`.

Listing 8.4 Command as a free monad

```
trait Commands[A] {
    type Command[A] = Free[Event, A]
}

trait AccountCommands extends Commands[Account] {
    import Event._
    import scala.language.implicitConversions

    private implicit def liftEvent[A](event: Event[A]): Command[A] =
        Free.liftF(event)

    def open(no: String, name: String,
            openingDate: Option[DateTime]): Command[Account] =
        Opened(no, name, openingDate, today)

    def close(no: String, closeDate: Option[DateTime]): Command[Account] =
        Closed(no, closeDate, today)

    def debit(no: String, amount: Amount): Command[Account] =
        Debited(no, amount, today)

    def credit(no: String, amount: Amount): Command[Account] =
        Credited(no, amount, today)
}
```

A Command is a free monad over Event.

This implicit function lifts an Event into the context of the free monad. See the text for more details of how this helps in building composite commands.

A sample Command. Note you emit an Event, which is a data type without any semantics or processing logic of what the command does. The implicit function liftEvent lifts this event, Opened, into the context of the free monad. Hence you can have as the return type Command[Account], which, when desugared, is Free[Event, Account] and which is the free monad.

Intuitively, listing 8.4 defines the commands such as `open` and `close`, and each maps to an `Event` type, which it will publish on successful execution when you allow for its interpretation. You also have an implicit function, `liftEvent`, that lifts the `Event` into the context of its free monad. This explains why each command becomes a free

monad of type `Command[Account]`. Figure 8.8 explains this transformation with a simple diagram.

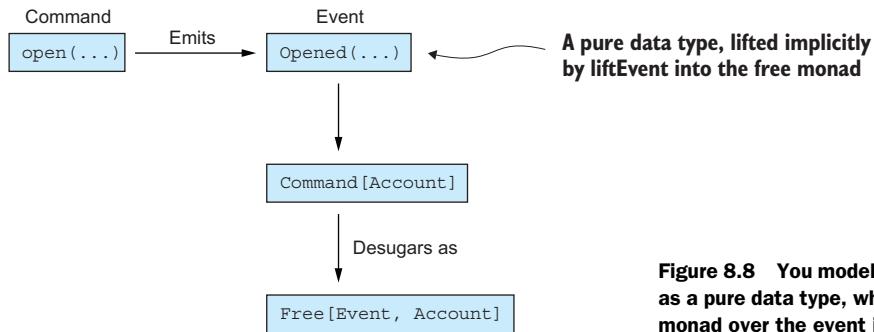


Figure 8.8 You model a command as a pure data type, which is a free monad over the event it publishes.

By virtue of being a monad, you can compose commands algebraically to form larger ones:

```

val composite =
  for {
    a <- open("a-123", "John J", Some(today))
    - <- credit(a.no, 10000)
    - <- credit(a.no, 30000)
    d <- debit(a.no, 23000)
  } yield d
  
```

Here `composite` is still a pure data type. You'll see in the next section how to peel off the layers of monads from a composite command, extract the events that hide under it, interpret the algebra of the events, and assign actions to each of them.

8.4.3 Interpreters—hideouts for all the interesting stuff

As experts of free monads, I'm sure you know by now that the ideal place to process commands with side effects is the interpreter of the free monad. You build your commands through composition and finally submit to the interpreter for doing the actual stuff. Let's have a simple interpreter that dispatches on the event that each of our commands publish and does all the necessary domain logic. The following listing provides one such interpreter.

Listing 8.5 Interpreter of the free monad

```

import scalaz._
Import Scalaz._
import scalaz.concurrent.Task
trait RepositoryBackedInterpreter {
  def step: Event ~> Task
  def apply[A](action: Free[Event, A]): Task[A] ←
    = action.foldMap(step)
}
  
```

The interpreter takes the entire free monad (the composite command), steps through the recursive structure of the composite, and at each step executes the `step` function. The `step` function is where the command handlers will be invoked and needs to be implemented separately for each aggregate type.

Let's talk a bit about how this interpreter works. The implementation looks too succinct, and that's because you haven't yet provided the details of the domain logic processing. But this is the kernel of how the interpretation of the free monad takes place, and if you understand these four lines of code, you should be able to appreciate the details. Here are the steps that this interpreter uses to process a command, which, however, can be composite:⁹

- `apply` is the entry point of the interpreter, which takes a free monad as its input. In this case, you can pass a command, which is `Free[Event, Account]`. Note that this free monad is a recursive data structure, so you can have layers of commands that need to be executed, one after the other (much like the composite example from the preceding section).
- The basic function of the `apply` method is to recurse through the entire stack of commands and return a `Task` that you can execute. The free monad implementation of Scalaz has a function, `foldMap`, that *folds over* the monad and maps the `step` function for each layer. The `step` function returns a `Task` that again is a monad.¹⁰ So all the `Tasks` that `foldMap` generates *flatten* together to form the final `Task` that `apply` returns.¹¹
- The `step` function returns a natural transformation, which, as you saw in chapter 5, is a structure-preserving transformation. Here you build a `Task` from an `Event`. Finally, the interpreter returns a `scalaz.concurrent.Task` that you can use with one of its supported strategies for evaluation.

THE INTERPRETER THAT DELIVERS ACCOUNT SERVICE

You're now into the final parts of our implementation, where all the domain behaviors need to execute, command handlers need to fire, side effects need to run, and events need to be generated. That sounds like a mouthful, but the implementation is straightforward. Until now, we've organized our code as pure modules; even commands are (until now) pure data types. You'll soon give semantics to these inert commands as part of our account-specific interpreter. The following listing gives an overview of structuring our interpreter. I won't give the whole implementation here; as usual, you can find a completely runnable version as part of the online code repository for the book.

⁹ Understanding exactly how the interpreter works requires knowledge of the way free monads are implemented in Scalaz. This is covered in chapter 5, section 5.5.5.

¹⁰ `Task` is a monad; see <https://github.com/scalaz/scalaz> for `Task` source code.

¹¹ Remember, a monad offers a `flatMap`, which is a `map`, followed by `flatten`. Here all `Tasks` that are generated `flatMap` to form the final `Task`.

Listing 8.6 The domain logic for processing commands

```

object RepositoryBackedAccountInterpreter extends
  RepositoryBackedInterpreter {
  import AccountSnapshot._

  val eventLog = InMemoryEventStore.apply[String] ← 1

  import eventLog._

  val step: Event ~> Task = new (Event ~> Task) {
    override def apply[A](action: Event[A]): Task[A] =
      handleCommand(action)
  }

  private def validateClose(no: String, cd: Option[DateTime]) = for {
    l <- events(no)
    s <- snapshot(l)
    a <- closed(s(no))
    _ <- beforeOpeningDate(a, cd)
  } yield s ← 2

  // all other domain validation functions ..

  private def handleCommand[A](e: Event[A]): Task[A] = e match { ← 3
    case o @ Opened(no, name, odate, _) => Task {
      validateOpen(no).fold(
        err => throw new RuntimeException(err),
        _ => {
          val a = Account(no, name, odate.get)
          eventLog.put(no, o)
          a
        }
      )
    }
    case c @ Closed(no, cdate, _) => Task {
      validateClose(no, cdate).fold(
        err => throw new RuntimeException(err),
        currentState => {
          eventLog.put(no, c)
          updateState(c, currentState)(no)
        }
      )
    }
  }
  // all other command handlers ..
}

```

Invokes the validation functions → 1

The event log where events get appended. This is an in-memory implementation. The next section covers the abstraction for the event store.

A domain validation function that's invoked as part of the close command. See text for details on validation processing.

Looks up the aggregate in the event log → 2

Prepares its snapshot → 3

The core function for command handling. Note how it dispatches on the event and invokes specific domain behaviors on the aggregate. → 4

The three main parts of `RepositoryBackedAccountInterpreter` are as follows:

- **Event log**—You define the event log here. In our use case, it's a simple in-memory Map (you can have a look at the source code in the repository). But you have a generic interface, and you can have multiple implementations for the same. In

fact, you have an implementation in the online repository that stores JSON instead of the event objects.

- *Domain validations*—An important aspect of command handlers is executing domain validations, and you define the validations as part of the interpreter. I've shown one validation function here just to give you an idea of how to use monadic effects to look up the aggregate from the event log ①, prepare its snapshot ②, and then invoke the validation functions on it ③.
- *Command handler*—All commands get executed as part of the interpretation of our event algebra. The `apply` method of `RepositoryBackedInterpreter` in listing 8.5 is the entry point of the command handler. It takes the whole command stack, recurses through the structure, and passes every event to the `handleCommand` function. Note the event that it passes is the one that the command used when it built up the free monad (① in listing 8.4). So you started with the command, built as a free monad over an event, composed a bunch of those to form composite commands, and now the command handler executes those commands by interpreting the events (④ in listing 8.6). It uses the event algebra and does a pattern match. Then the validations are done, and the handler writes the event into the event log. Finally, the command handler uses the `updateState` API to update the state of the aggregate (`Account`) by applying the current event to it.

A SAMPLE RUN

Let's tie all the strings together and see what a sample run looks like. Listing 8.7 has been manually formatted to fit these pages, but you'll get a similar result when you run the code from the book's online repository. This section has covered parts of the implementation to highlight the functional aspects of the way it works. You can get the complete runnable implementation from the online repository.

In the online repository, you'll see a few more abstractions that make the implementation modular and reusable. You'll see the implementation split into generic components that can be reused across any aggregate. And you'll have some account-specific logic as well (for example, the specific commands and events that apply only to an account).

Listing 8.7 A sample run

```
scala> import frdomain.ch8.cqrs.service._  
scala> import Scripts._  
scala> import RepositoryBackedAccountInterpreter._  
  
scala> RepositoryBackedAccountInterpreter(composite)  
res0: scalaz.concurrent.Task[Account] = scalaz.concurrent.Task@2aaa2ff0
```

←

A sample composite command, which is a free monad being interpreted by your interpreter generating a Task

```

scala> res0.unsafePerformAsync
res1: Account = Account(a-123,debasish ghosh,2015-11-
    22T12:00:09.000+05:30,None,Balance(17000)) ←
    | Runs the task to create a
    | snapshot of the account that the
    | composite command created
    | through some debits and credits

scala> for {
    |   a <- credit("a-123", 1000)
    |   b <- open("a-124", "john j", Some(org.joda.time.DateTime.now()))
    |   c <- credit(b.no, 1200)
    | } yield c
res2: scalaz.Free[Event,Account] = Gosub(..)

scala> RepositoryBackedAccountInterpreter(res2)
res3: scalaz.concurrent.Task[Account] = scalaz.concurrent.Task@3bb6d65c

scala> res3.unsafePerformSync
res4: Account = Account(a-124,john j,2015-11-
    22T12:01:23.000+05:30,None,Balance(1200)) ←
    | Creates another composite
    | command that creates another
    | account, does some operations
    | on it, and also on the account
    | created in the preceding step

scala> eventLog.events("a-123")
res5: scalaz.\/[Error,List[Event[_]]] = \/- (List(Credited(a-123,1000,
    ↵ 2015-11-22T12:00:09.000+05:30,<function1>), Debited(a-123,23000,
    ↵ 2015-11-22T12:00:09.000+05:30,<function1>), ...)

scala> eventLog.events("a-124")
res6: scalaz.\/[Error,List[Event[_]]] = \/- (List(Credited(a-124,1200,
    ↵ 2015-11-22T12:00:09.000+05:30,<function1>), Opened(a-124,
    ↵ john j,Some(2015-11-22T12:01:23.000+05:30),2015-11-
    22T12:00:09.000+05:30,<function1>))) ←
    | Fetches the set of events for both
    | the accounts from the event log

scala> import AccountSnapshot._ ←
import AccountSnapshot._

scala> import scalaz._ ←
import scalaz._

scala> import Scalaz._ ←
import Scalaz._

scala> res5 |+| res6
res7: scalaz.\/[Error,List[Event[_]]] = \/- (List(Credited(
    ↵ a-123,1000,2015-11-22T12:00:09.000+05:30,<function1>),
    ↵ Debited(a-123,23000,2015-11-22T12:00:09.000+05:30,<function1>), ...) ←
    | Note the return type of events is a scalaz.\/
    | that's a monoid. You can combine them with
    | monoid append and get the total set of events.

scala> res7 map snapshot
res8: scalaz.\/[Error,scalaz.\/[String,Map[String,Account]]] =
    \/- (\/- (Map(a-124 -> Account(a-124,john j,2015-11-
        22T12:01:23.000+05:30,None,Balance(1200)), a-123 -> Account(a-123,
        ↵ debasish ghosh,2015-11-22T12:00:09.000+05:30,None,
        ↵ Balance(18000)))) ←
    | Now you can run snapshot on this
    | set of events and get the latest
    | snapshot for both the accounts.

scala> eventLog.allEvents
res9: scalaz.\/[Error,List[Event[_]]] = \/- (List(Credited(a-123,
    ↵ 1000,2015-11-22T12:00:09.000+05:30,<function1>),
    ↵ Debited(a-123,23000,2015-11-22T12:00:09.000+05:30,<function1>), ...)

```

```
scala> res9 map snapshot
res10: scalaz.\/[Error,scalaz.\/[String,Map[String,Account]]] = 
  ↪ \/-(\/-(Map(a-124 -> Account(a-124,john j,
  ↪ 2015-11-22T12:01:23.000+05:30,None,Balance(1200)),
  ↪ a-123 -> Account(a-123,debasish ghosh,
  ↪ 2015-11-22T12:00:09.000+05:30,None,Balance(18000))))
```

You can also fetch all events from the event log at once and run a snapshot on the full set. This is useful when you want to build your aggregate snapshot from scratch using the event stream from event log.

8.4.4 **Projections—the read side model**

So far, you've seen how to process commands and publish events that get appended to an event log. But what about queries and reports that your model needs to serve? Section 8.3 introduced read models designed specifically for this purpose. Read models are also called *projections*, which are essentially mappings from the write model (event log) into forms that are easier to serve as queries. For example, if you serve your queries from a relational database, the table structures will be suitably denormalized so that you can avoid joins and other expensive operations and render queries straight-away from that model of data.

Setting up a projection is nothing more than having an appropriate snapshotting function that reads a stream of events and updates the current model. This update needs to be done for the new events that get generated in the event log. You need to consider a few aspects of projection architecture before deciding on how you'd like to have your read models:

- *Push or pull*—You can have the write side push the new events to update the read model. Alternatively, the read model can pull at specific intervals, checking whether any new events are there for consumption. Both models have advantages and disadvantages. The pull model may be wasteful if the rate of event generation is slow and intermittent; the read side may consume lots of cycles pulling from empty streams. The push model is efficient in this respect: Events get pushed only when they're generated. But pull models have built-in back-pressure handling; the read side pulls, depending on the rate at which it can consume. Push models need an explicit back-pressure handling mechanism, as you saw with Akka Streams in chapter 7.
- *Restarting projections*—In some cases, you may need to change your read model schema, maybe because of a change in the event structure in the write model. In such a situation, you need to migrate to the updated version of the model with minimal impact on the query/reporting service. This is a tricky situation, and the exact strategy may depend on the size and volume of your read model data. One strategy is to start hydrating a new projection from the event log while still serving queries from the older one. After the new projection model catches up with all events from the write model, you switch to serving from the new one.

8.4.5 The event store

The *event store* is the core storage medium for all domain events that have been created in your system to date. You need to give lots of thought to selecting an appropriate store that meets the criteria of scalability and performance for your model. Fortunately, this storage needs much simpler semantics than for a relational database. This is because the only write operation that you want to do is an append. Many append-only stores are available that offer scalability, including Event Store (<https://geteventstore.com>) or some of the NoSQL stores such as Redis (<http://redis.io>) or Cassandra (<http://cassandra.apache.org>).

The nature of storage in an event store is simple. You need to store events indexed by aggregate ID. You should have at least the following operations from such a store:

- Get a list of events for a specific aggregate.
- Put an event for a specific aggregate into the store.
- Get all events from the event store.

The following listing shows the general contract for such an event store.

Listing 8.8 A simple contract for an event store

```
import scalaz.\/
trait EventStore[K] {
  def get(key: K): List[Event[_]]
  def put(key: K, event: Event[_]): Error \/ Event[_]
  def events(key: K): Error \/ List[Event[_]]
  def allEvents: Error \/ List[Event[_]]
}
```

The online repository for this book contains a couple of implementations for an event store, which you can go through. The event store is one of the central artifacts that can make or break the reliability of your model. Our examples have used a simple implementation based on a thread-safe concurrent Map. But in reality, you need to choose implementations that guarantee reliability as well as high availability. In most production-ready implementations, writes are fsynced to a specific number of drives by using quorums before you declare the data to have been safely persisted. Some implementations also offer ACID semantics over transactions in event stores. You can have long-lived transactions and treat the whole as an atomic one—just as in an RDBMS.

8.4.6 Distributed CQRS—a short note

Event sourcing and CQRS rely on event recording and replays for managing application state. Event stores typically need to be replicated across multiple locations, or nodes across a single location, or even processes across a single node. Also, you may have multiple event stores, each having different event logs collaborating together through replication to form a composite application state. All replication has to be asynchronous. Hence you need a strategy for resolving conflicting updates. If you have a domain model that needs to implement distributed domain services that rely

on multiple event stores collaborating toward consistency, be sure to pick up CQRS frameworks that offer such capabilities.

These frameworks replicate events across locations, using protocols that preserve the happened-before relationship (causality) between events. Typically, they use a vector-clock-based implementation to ensure tracking of causality. One such framework is Eventuate (<http://rbmhtechology.github.io/eventuate>), which is based on Akka Persistence's event-sourced actor implementation (<http://doc.akka.io/docs/akka/2.4.4/scala/persistence.html>). Eventuate uses event-sourced actors for command handling and event-sourced views and writers for queries (projection model). Using event-sourced processors, you can build event-processing pipelines in Eventuate that can form the backbone of your distributed CQRS implementation.

This section is a reference for implementing distributed CQRS-based domain services. You won't implement the details of the service. You can look at frameworks such as Eventuate if you need an implementation, but in most cases you may not need distribution of services. And this is true even if your model has multiple bounded contexts that are decoupled in space and time. Keep data stores local to your bounded context and ensure that services in each bounded context access data only from that store. Design each bounded context as an independently managed and deployable unit. This design technique has a popular name today: *microservices*. Chris Richardson has a nice collection of patterns that you can follow to design microservice-based architectures (<http://microservices.io/patterns/microservices.html>).

One of the most important consequences of this paradigm is that you have a clear separation between bounded contexts and don't have to manage causality of events across contexts. It's also not mandatory that you implement CQRS in all of the bounded contexts. Each context may have its own technique of data management. But because a decoupling between contexts occurs, you never have to deal with resolution of conflicts in data consistency across cross-context services. The net result is that within each bounded context where you decide to implement CQRS, you can follow the purely functional implementation that you saw in this chapter.

8.4.7 **Summary of the implementation**

It has been a long story so far, with the complete implementation of the event-sourced version of persisting aggregates. You didn't try a generic framework, but instead followed a specific use case to see the step-by-step evolution of the model. Let's go through a summary of the overall implementation so that you get a complete picture of how to approach the problem for your specific use case:

- 1 Define an algebra for the events. Use an algebraic data type and have specialized data constructors for each type of event.
- 2 Define the command as a free monad over the event. This makes commands composable. You can build composite commands through a for-comprehension with individual commands. The resultant command is still just a data type without

any associated semantics. Hence a command is a pure abstraction under our model of construction.

- 3 Use the algebra of events in the interpreter to do command processing. This way, you have all effects within the interpreter, keeping the core abstraction of a command pure. On successful processing, commands publish events as part of the interpretation logic, which get appended to the event log.
- 4 Define a suitable read model (also known as a projection), and use an appropriate strategy to hydrate the read model from the event stream on the write side.

8.5 Other models of persistence

CQRS and event sourcing offer a persistence model that provides a functional implementation. Events are nothing more than serialized functions, descriptions of something that has already happened. It's no coincidence that many of the implementations of event sourcing give a functional interface of interaction. But the paradigm has quite a bit of complexity in architecture that you can't ignore:

- *Different paradigm*—It's a completely different way to work with your data layer, miles away from what you've been doing so far with your RDBMS-based application. You can't ignore the fact that it's not familiar territory to most data architects.
- *Operational complexity*—With separate write and read models, you need to manage more moving parts. This adds to the operational complexity of the architecture.
- *Versioning*—With a long-lived domain model, you're bound to have changes in model artifacts. This will lead to changes in event structures over time. You have to manage this with event versioning. Many approaches to this exist, and none of them is easy. If you plan to have an event-sourced data model for your application, plan for versioned events from day one.

In view of these issues, it's always useful to keep in mind that you can use a relational model for your data in a functional way with your domain model. You no longer need to use an object relational framework that mandates mutability in your model. We're seeing libraries that help you do so, using the benefits of immutable algebraic data types and functional combinators along with your domain APIs to interact with an underlying RDBMS. This section presents a brief overview of Slick, an open source, functional relational mapping library for Scala. But this isn't a detailed description of how Slick works. For more details, see the Slick website (<http://slick.typesafe.com>) or the excellent book *Essential Slick* by Richard Dallaway and Jonathan Ferguson (<http://underscore.io/books/essential-slick/>).

8.5.1 Mapping aggregates as ADTs to the relational tables

Continuing with our example of the Account aggregate, listing 8.9 defines a composite aggregate for a customer account and all of its balances across a period of time.

Note the Balance type contains a field, `asOnDate`, that specifies the date on which the balance amount is recorded.

Listing 8.9 Account and Balance aggregates

```
import java.sql.Timestamp

case class Account(id: Option[Long],
  no: String,
  name: String,
  address: String,
  dateOfOpening: Timestamp,
  dateOfClosing: Option[Timestamp]
)
case class Balance(id: Option[Long],
  account: Long, ←
  asOnDate: Timestamp,
  amount: BigDecimal
)
```

Note you keep a reference to the account ID in the Balance aggregate.

As you can see, this is a shallow aggregate design. Instead of storing a reference to an Account within Balance, you store an account ID. This is often considered a good practice when you're dealing with a relational model underneath, because it keeps your domain model aggregate closer to the underlying relational structure. *Effective Aggregate Design*, an excellent series of articles by Vaughn Vernon, presents some of the best practices in designing aggregates (see <https://vaughnvernon.co/?p=838>).

Using Slick, you can map your aggregate structure to the underlying relational mode. The following listing shows how to do this and how to define your table structure as an internal DSL in Scala.

Listing 8.10 The table definitions in Scala DSL using Slick

```
import Accounts._ ←
import Balances._ ←
class Accounts(tag: Tag) extends Table[Account](tag, "accounts") { ←
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc) ←
  def no = column[String]("no") ←
  def name = column[String]("name") ←
  def address = column[String]("address") ←
  def dateOfOpening = column[Timestamp]("date_of_opening") ←
  def dateOfClosing = column[Option[Timestamp]]("date_of_closing") ←
  def * = (id.?, no, name, address, dateOfOpening, dateOfClosing) ←
    <> (Account.tupled, Account.unapply) ←
  def noIdx = index("idx_no", no, unique = true) ←
}
object Accounts {
  val accounts = TableQuery[Accounts]
}
```

Note how you map the ADT Account to the table definition.

You define a primary key for an account as an auto-increment field.

Defines the projection for query—maps to the Account ADT

```

class Balances(tag: Tag) extends Table[Balance](tag, "balances") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def account = column[Long]("account")
    def asOnDate = column[Timestamp]("as_on_date")
    def amount = column[BigDecimal]("amount")

    def * = (id.?, account, asOnDate, amount)
        <-- (Balance.tupled, Balance.unapply)
    def accountfk = foreignKey("ACCOUNT_FK", account, accounts)
        (_._id, onUpdate=ForeignKeyAction.Cascade,
         onDelete=ForeignKeyAction.Cascade)
}
object Balances {
    val balances = TableQuery[Balances]
}

```

Defines a foreign key that links Account and Balance

This maps our ADTs `Account` and `Balance` with the underlying table structures by defining them as shapes that can be produced through data manipulation with the underlying database. Slick does this mapping through a well-engineered structure within its core. The net value that you gain is a seamless interoperability between the database structure and our immutable algebra of types. You'll see this in action when we discuss how to manipulate data using the functional combinators of Slick.

8.5.2 Manipulating data (functionally)

As a library in Scala, Slick offers functional combinators for manipulating data that look a lot like the Scala Collections API. This makes using the Slick APIs more comfortable for a Scala user. Consider the example in the following listing; you'd like to query from your database, using the schema defined in listing 8.10, the collection of `Balance` records for a specific `Account` within a specified time interval.

Listing 8.11 Balance query using functional combinators of Slick

db is the database handle, and you ask the balance for accountNo within the date range fromDate & toDate.

```

def getBalance(db: Database, accountNo: String, fromDate: Timestamp,
              toDate: Timestamp) = {
    val action = for {
        a <- accounts.filter(_.no === accountNo)
        b <- balances if a.id === b.account &&
                    b.asOnDate >= fromDate && b.asOnDate <= toDate
    } yield b
    db.run(action.result.asTry)
}

```

db.run returns a Future, which you can compose in a nonblocking way.

Familiar comprehension syntax just as in Scala collections, only over an underlying database

As you can see, the core query API resembles the same model as the Scala collections. The query `accounts.filter` fetches a projection from the `Accounts` table and binds monadically with the query that works on the `Balances` table through the

for-comprehension. This is, in effect, a relational join implemented monadically. For more variants of joins and queries, refer to the documentation on Slick (<http://slick.typesafe.com>). The query returns a `Future`, which, as you know by now is one of the core substrates that make your model reactive. Instead of waiting on the `Future`, you can compose with other APIs that return `Futures` and build larger, nonblocking APIs. But Slick offers more-reactive APIs by allowing you to stream your database query results directly into an Akka Streams-based pipeline. You'll take a brief look at this capability next.

8.5.3 Reactive fetch that pipelines to Akka Streams

Consider a use case for fetching a large volume of data from your database and processing it as part of your domain logic. In order to be reactive with bounded latency and guaranteed response time from the server, you can use the power of streams. And reactive frameworks such as Slick allow you to directly publish your database query result as a source into your Akka Streams pipeline (<http://doc.akka.io/docs/akka/2.4.4/scala/stream/index.html>).

Here's a simple query that fetches the sum of all balances grouped by accounts within a specified time period. This is bound to be a huge result set for a nontrivial database of any financial company that serves retail customers. The following listing shows the query implemented using Slick.

Listing 8.12 Balance query that generates a stream of result

```
type BalanceRecord = (String, Timestamp, Timestamp, Option[BigDecimal])
def getTotalBalanceByAccount(db: Database, fromDate: Timestamp,
  toDate: Timestamp): DatabasePublisher[BalanceRecord] = {
  val action = (for {
    a <- accounts
    b <- balances if a.id === b.account && b.asOnDate >= fromDate
    && b.asOnDate <= toDate
  } yield (a.no, b)).groupBy(_.no).map { case (no, bs) =>
  (no, fromDate, toDate, bs.map(_.amount).sum)
}
db.stream(action.result)
```

This listing has a couple of important takeaways for accessing your database in a reactive way:

- *Back-end query power*—Slick offers lots of combinators to process data at the server level and can generate optimized SQL for them. In listing 8.12, use of `groupBy` is a similar example. By using `groupBy`, you can do the grouping at the server level only (much as you would with SQL) so that you save on expensive data transport to do the formatting on the client side.

- *Reactive streams integration*—The query returns a `DatabasePublisher`, which you can hook on straightaway to an Akka Streams flow graph as a `Source`; for example, `val accountSource: Source[BalanceRecord] = Source(getTotalBalanceByAccount(...))`.

In summary, you can use a relational database with an appropriate functional library to model your database as a reactive persistence layer.

8.6 Summary

This chapter was an introduction to persisting domain models in an underlying database. Gone are the days when we assumed that we'd have a relational database for storing our domain-model elements. New paradigms have emerged, and we've learned to better align our functional domain model with underlying storage that also values immutability of data. We discussed event sourcing and CQRS that eschew the concept of in-place updates of data and instead store your domain model as an immutable stream of events. The main takeaways of this chapter are as follows:

- *CRUD isn't the only model of persistence*—We discussed the rationale for using events as the source of ground truth and introduced event sourcing as one technique to store domain events. This makes our model more auditable and traceable and leads to a huge increase in the perceived business value of the underlying storage. It's not only storage of relational data; it's now storage of domain events as well.
- *Implementing an event-sourced domain model functionally*—We demonstrated how to implement an event-sourced model using functional techniques. Lots of other implementations are available today. But the approach we took goes well with the principles of functional programming. Free monads give you a pure implementation of commands and event algebras, whereas interpreters deal with side effects.
- *FRM, not ORM*—If you decide to use the CRUD model for persistence (and there are valid reasons to do so), use a functional relational mapping framework such as Slick. We discussed in brief how to do so with a purely functional interface to the relational database underneath.



Testing your domain model

This chapter covers

- Designing testable domain models
- Learning why xUnit-based testing isn't enough
- Introducing property-based testing
- Using properties as executable blueprints for domain behaviors

This chapter covers testing your domain model. I begin with an explanation of what is meant by a testable model and how testability is related to the modularity of a model architecture. I take examples from some of the model components discussed in earlier chapters and look at how to test them. Then I discuss briefly the shortcomings of xUnit-based testing and dive into the details of property-based testing using ScalaCheck.

Figure 9.1 diagrams the chapter content.

9.1 Testing your domain model

Before delving into testing domain models, you first need to understand what exactly I mean by *testability*. After all, a group of team members manually harnessing your domain model with reams of test data is also doing testing. But that's not the

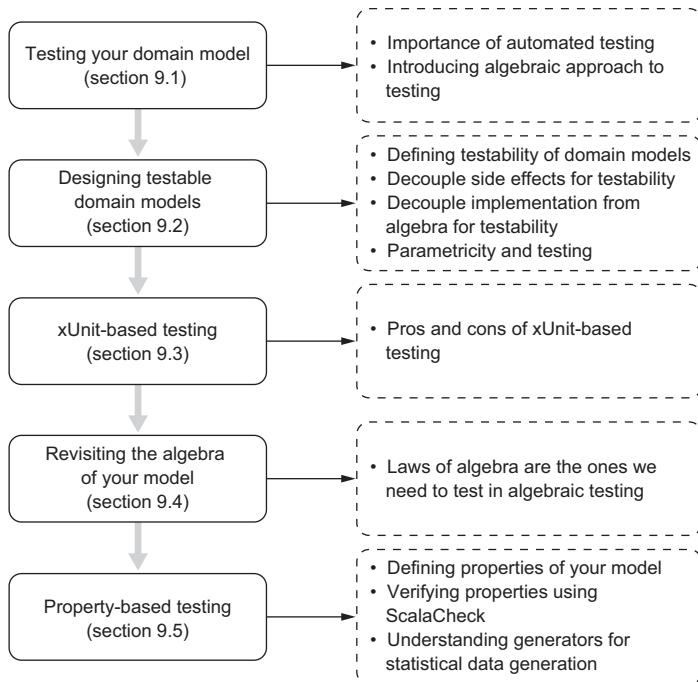


Figure 9.1 The progression of this chapter's sections

definition of testing that we're using here. The main problem with manual testing is that it's not repeatable. After you make changes to your domain-model implementation, how do you ensure that it still works as per the specification? Would you employ the same set of testing team members and repeat the entire ceremony? How do you find out which parts of the model have been impacted by the changes that you made last night? Manual testing is *no* testing.

You need automated testing that is

- Repeatable
- Scalable with an additional dataset
- Maintainable
- Noninvasively extensible with addition of features
- Capable of being integrated with the build system

Tests can be at various levels of granularity. You can have unit tests, module tests, and integration tests, for example, and we won't spend too much time talking about how they differ. The type of testing that we focus on in this chapter is white-box testing: You write tests at the function level and verify that the function honors all the promises that it makes. In previous chapters, we talked a lot about the algebra of a function. In this chapter, you'll see how to use that algebra to verify the properties that a

function satisfies. A function embodies a set of domain behaviors that you can express as a collection of algebraic properties. As an example from our banking domain, a withdrawal (or debit) from a bank account followed by a deposit (or credit) of an equal amount restores the original balance of the account. This is an algebraic property that domain behaviors such as *debit* and *credit* have to honor. Our approach to testing focuses on verifying those algebraic properties. Remember that when we're talking about functions, we're talking about *pure* functions. So verifying algebraic properties of pure functions is repeatable, deterministic, and scalable with additional datasets. You'll learn the exact techniques for testing algebraic properties of functions later in this chapter.

But first you need to understand what it means to make a model *testable*.

9.2 Designing testable domain models

An abstraction is testable if it's *testable in isolation*. You should be able to test the abstraction without testing the other abstractions that it collaborates with. Let's start with an example of an initial implementation of a function that opens a customer account with a bank. The following listing starts with a simple implementation for Account and focuses on the behavioral aspects of the open function.

Listing 9.1 Toward testable domain models

```
case class Account(no: String, name: String, idNo: String,
  dateOpened: DateTime, dateClosed: Option[DateTime])
```

```
trait AccountService {
  type Error = String
  type ErrorOr[A] = Error \/
    Account
```

```
def open(no: String, name: String, idNo: String,
  dateOpened: DateTime): ErrorOr[Account] = {
  val isValid: Boolean = verifyId(idNo, name)
  if (isValid) {
    //... other validations
    Account(no, name, idNo, dateOpened, None).right
  } else s"Id ($idNo) validation failed".left
}
```

```
def verifyId(idNo: String, name: String): Boolean = {
  // possibly an expensive call that needs to communicate
  // with external systems, maybe over a Web service
  // stubbed here
  true
}
```

How easily testable is the open function? At the beginning, it calls the function verifyId that does the actual verification of the customer's ID and name. Though you've stubbed out the implementation of verifyId, in reality it may need to reach out to a third-party implementation, possibly through a web service, before returning the

result of the verification. It's a call that induces a side effect in `open` and is considered impure in our vocabulary of pure functional programming. Admittedly, it's difficult to test `open` with such an external service call in between. But more than that, testing `verifyId` for correctness isn't the responsibility of the suite that tests the correctness of the `open` function.

You need to find a way to move `verifyId` out of your test path for `open`. One common practice is to use a mocking library that mocks out external dependencies by supplying fake instances and allowing developers to verify the system under test. Besides introducing an external dependency in the form of yet another library, mocks introduce lots of boilerplates. They're also often tightly coupled with domain-model objects and hence need to be reengineered in the event your domain model changes.

With functional programming, you can use the power of functions to mock implementations out without the additional complexities of introducing yet another external library in your stack of development.

9.2.1 Decoupling side effects

To make the `open` function more testable, you need to decouple the side effect of `verifyId` from `open`. You learned the trick of dependency injection earlier in the book.¹ You'll use the same trick here: Make `verifyId` part of an abstraction that you can inject from outside. The moment you make `verifyId` part of the environment, you gain the flexibility of injecting an instance that's pure and that bypasses the entire logic of the external service call. And as you remember from injecting repositories into our domain services earlier in the book, the first step is to make `open` return a function. The following listing does the same.

Listing 9.2 Toward testable domain models

```
case class Account(no: String, name: String, idNo: String,
  dateOpened: DateTime, dateClosed: Option[DateTime])

trait IdVerifier {
  def verifyId(idNo: String, name: String): Boolean
}

trait AccountService {
  type Error = String
  type ErrorOr[A] = Error \/

  def open(no: String, name: String, idNo: String,
    dateOpened: DateTime)
    : IdVerifier => ErrorOr[Account] = { (v: IdVerifier) =>
    if (v.verifyId(idNo, name)) {
      //... other validations
      Account(no, name, idNo, dateOpened, None).right
    } else Left(s"Id $idNo is not valid")
  }
}
```

¹ I discussed how to control the instance of repositories being injected into services in section 3.3.6. If you need a refresher, take a look at that section before you proceed.

```

    } else s"$Id ($idNo) validation failed".left
}
}

case class MockIdVerifier() extends IdVerifier {
  def verifyId(idNo: String, name: String) = true
}

```

1 Mocked implementation of verifyId for testing

Here, `open` returns a function that takes an argument, `IdVerifier`, which is the module that provides the algebra of `verifyId`. You can have multiple implementations of the module: one for the production system that uses the actual version that interacts with the external service call, and another mocked implementation for the purpose of testing ①. In summary, you've made the function more testable by decoupling it from the implementation of an impure function.

9.2.2 Providing custom interpreters for domain algebra

You just saw an instance of a single function `purified` for enhanced testability. You can take this further up in your level of abstraction and provide custom pure implementations for an entire algebra. You already saw this technique in chapter 5, when we discussed free monads. Let's revisit this topic once again in the context of domain model testing.

In section 5.5, listing 5.6, you designed an algebra for an account repository that handles the interaction of accounts with the underlying data layer.² Here's the algebra you defined:

```

sealed trait AccountRepoF[+A]
case class Query(no: String) extends AccountRepoF[Account]
case class Store(account: Account) extends AccountRepoF[Unit]
case class Delete(no: String) extends AccountRepoF[Unit]

```

The base trait for actions

Every action expressed as an ADT (pure data)

And you defined a free monad over the base trait `AccountRepoF`:

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

What `Free` gives you is a monad for the algebra. You can now use the various elements of the algebra and compose them monadically to build larger abstractions. For example, you can compose `Query` and `Store` to implement the update functionality.³ The algebra is a pure abstraction without any semantics of what `Query`, `Store`, and `Delete` do. It's the interpreter of the algebra that implements these semantics. So the technique that helps design testable APIs is to make your API depend *only* on the algebra and have the interpreter pluggable from the environment (just as you did for `IdVerifier` in section 9.2.1).

² If you missed this section, this may be a good time to visit it.

³ If you need to refresh the exact implementation, listing 5.8 gives a few examples of how to achieve this.

In a production system, you'll typically implement a repository using an enterprise-scale database. But for testing, you should be able to test your APIs with a simpler alternative implementation such as a `Map[K, V]`. A free-monad-based implementation gives you exactly this facility. You can have separate interpreters: one for production use that implements semantics based on the enterprise database, and one for testing that uses an underlying `Map` as the storage data structure. Chapter 5 presented this technique and implemented an interpreter for `AccountRepository` based on a `Map` in listing 5.8.

Here's an example from the implementation of section 5.5. Using the algebra, you define a composite sequence of actions on a customer account. This is just pure data types composed together without any explicit behavior associated with `open`, `credit`, `debit`, or `query`:⁴

```
import AccountService._

val composite = for {
  x <- open("a-123", "debasish ghosh", Some(today))
  _ <- credit(x.no, 10000)
  _ <- credit(x.no, 30000)
  _ <- debit(x.no, 23000)
  a <- query(x.no)
} yield a
```

Now that you have your API for the composed action, you can use the interpreter as the last step of your computation. The interpreter accepts the algebra and injects the semantics by peeling off the layers of the free monad and interpreting the algebra of the functor inside. You can use either the `Map`-based interpreter for testing, `AccountRepoMapInterpreter().apply(composite)`, or use the database-based one in production—for example, `AccountRepoOracleInterpreter().apply(composite)`. In summary, a free-monad-based implementation provides excellent support for making your APIs testable.

9.2.3 **Implementing parametricity and testing**

One of the ways to make your domain model easily testable is to do away with writing explicit tests for some parts of your code: You can rely on someone else to do that for you. This section shows how to make the compiler do some testing for your model, thereby reducing the number of tests that you have to write. This isn't a new concept. You've already seen this technique in previous parts of this book, and we discussed this in detail in section 4.3. It's called *parametricity*; a parametric function is polymorphic on one or more types and is implemented only in terms of the algebra that those types have. For example, if you have a function `def accumulate[A: Monoid](a1: A, a2: A)`, the implementation of the function will only be in terms of the algebra of a

⁴ All these are operations implemented in chapter 5, section 5.5, using the algebra of the repository. See section 5.5 for details.

Monoid and nothing else. In that case, you call `accumulate` on a function parameterized on a Monoid.

When I discussed monoids in chapter 4, you saw how to implement parametric domain models by abstracting over domain behaviors and domain context. Replacing type-specific code with generic code proved to be more abstract and hence more reusable across various contexts. To refresh your memory, you implemented a generic combinator, `mapReduce`, in section 4.1.2 as follows:

```
def mapReduce[F[_], A, B](as: F[A])(f: A => B)
  (implicit fd: Foldable[F], m: Monoid[B]) = fd.foldMap(as)(f)
```

Note how you abstract an operation with `f: A => B` and the context on which to apply the operation using the type constructor `F`. In the context of our domain model, `f` can be any domain behavior, whereas `F` can be a collection of domain elements fetched from the database.

The primary benefit of these parametric functions is that you can test them using *any* type that satisfies the algebra, and that need not be complex types from the domain. In the preceding example, you can test `mapReduce` by using the identity function for `f` and a list of integers for `F[A]`. More important, you can define generic algebraic properties for parametric functions and use them to verify correctness. This makes them testable with any type that satisfies the algebra. The compiler ensures that you pass the correct types, and the algebraic properties ensure that it works correctly. Just imagine the amount of test code that you won't have to write by replacing all instances of such domain-specific types and operations with a small fragment of generic code parameterized by the appropriate algebra of types.

9.3 xUnit-based testing

The most common way to test a model today is to use xUnit-based testing methods. Frameworks such as JUnit (www.junit.org) and NUnit (www.nunit.org) provide abstractions that let you test *units* of your model in isolation. You can define your own unit (for example, with class-based, object-oriented methods, you usually define a class as a unit). Then you write assertions that validate the various methods of the class over the set of data that you provide.

After identifying the unit to test, you follow these main steps while writing an xUnit-based test:

- 1 Identify the method (function) to test.
- 2 Identify the scenario from the method. For example, you may want to write a scenario for a happy path for the method: All validations pass through, and you have a successful return from the method.
- 3 Provide a handcrafted dataset to test the scenario.
- 4 Check the assertion.

This methodology falls short in the third step, as you test the scenario with one or two pieces of data that you prepare up front. Often this isn't sufficient, because a manually

prepared dataset is never exhaustive from a business point of view. Often you'll miss edge cases or boundary conditions that may arise in production deployments. You'll see in the next section how to address this by using property-based testing.

9.4 Revisiting the algebra of your model

Chapter 3 indicated that the algebra of your model is composed of the following:

- The types that model domain entities, value objects, and so forth
- The functions that operate on types, with related functions grouped together as modules
- The laws or the business rules

In all the previous chapters, you saw how the first two artifacts manifest as part of the domain model. We discussed various ways to design typed domain models and to decompose domain behaviors into functions that operate on types. We're now ready to formalize the third component of the domain algebra: verifying the laws through the algebra of business rules. Note that these are verifications of business rules and not mere documentation. All the properties that you write demonstrate the correctness of business rules that govern the algebra of your domain model. Along with the types and functions, these properties form an equally important artifact of your model. So, treat them with equal care when designing your suite of properties. Figure 9.2 gives an

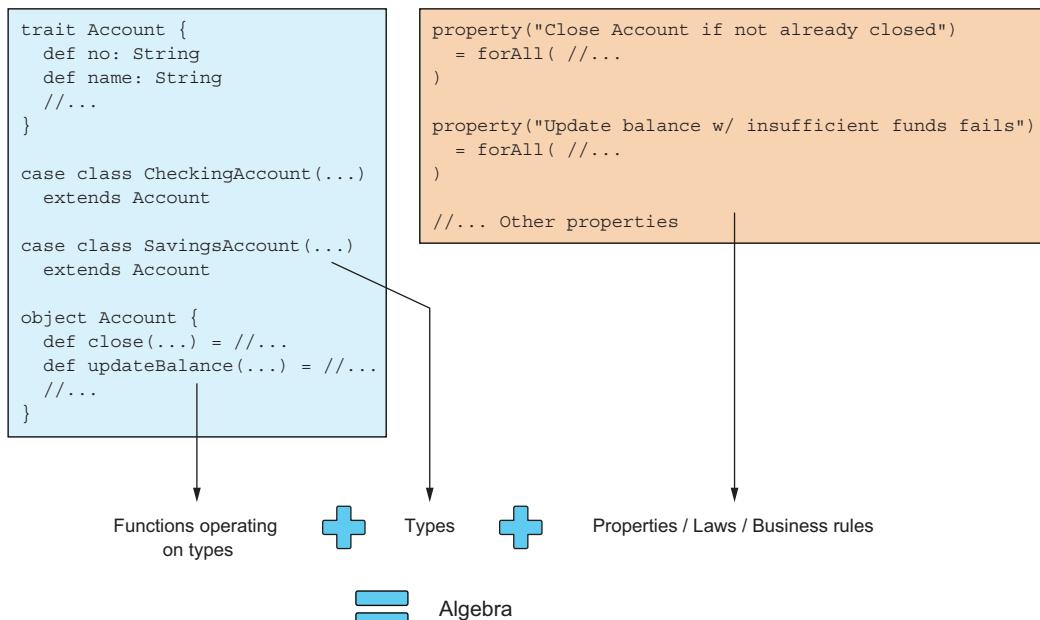


Figure 9.2 The complete domain algebra defined as the unification of three components: the types, the functions operating on types, and the business rules

overview of how the three components form the algebra of your domain. And the next section discusses all the implementation details of formulating properties as a verification tool for your domain behaviors.

9.5 **Property-based testing**

One of the core themes of this book so far has been the emphasis on the algebra of domain modeling. I like to think of a model as a union of algebras, with every collection of algebra delineated by separate bounded contexts. Within a bounded context, a specific type models a particular domain concept. You've seen types such as `Account`, `Customer`, `Balance`, and so forth named after the respective entities from the ubiquitous language of the domain. Each type participates in modeling domain behaviors; you grouped closely related behaviors into separate modules. Every module is a collection of functions that operate on types and models a specific algebra. These types don't hang around in isolation. The algebra that they model is often formed as a composition of other algebras. In chapter 4, you saw how to derive an implementation of the algorithm of trade generation by composing the algebras of domain-specific types such as `Trade`, `Order`, `Account`, and `Market` with generic algebraic structures including `Monad` and `Kleisli`.

When you compose algebras, you get behaviors that are algebraic as well. You can treat these behaviors as having algebraic properties based on the rules of the domain. As an example, when you model a transfer of funds between two customer accounts, the domain rule says that the amount that gets debited from one account must be equal to the amount that gets credited to the other. This is a domain rule, and any implementation that you have has to honor this, regardless of the accounts involved or the currency being transacted. You can think of this as an algebraic property that can be verified as a correctness criterion of a successful transfer operation. This is the essence of *property-based testing*. You define properties for your model and encode them such that they can be verified anytime you want.

Encoding algebraic properties for model elements and verifying them is a powerful way to test your system. We discuss this in more detail in this and some of the upcoming sections. By the end of this chapter, you'll see the usefulness of this technique and how it provides a better and more scalable model of testing than traditional xUnit-based ones. You'll use `ScalaCheck` (www.scalacheck.org) as the library for property-based testing. I introduce some of the concepts in the next sections and explain how they're implemented in `ScalaCheck`.

9.5.1 **Modeling properties**

You must now have an idea of what I mean by a *property* of a model element. A property abstracts an invariant predicate that holds true for a specific domain behavior. Let's look at an example that models one such invariant and implement a property using `ScalaCheck`:

```

property("Equal debit & credit retains the same position") =
  forAll(accountGen, moneyGen)(a: Account, m: Money) => {
    val Success((before, after)) = for {
      p <- position(a)
      r <- credit(p, m)
      q <- debit(r, m)
    } yield (p, q)

    before == after
  }

```

We haven't defined any of the domain types or functions (such as `position`, `debit`, and `credit`) used in the preceding code. But I'm sure they're intuitive and you can readily figure out what they do in the context of our domain of personal banking.⁵ Let's take some time to dissect each interesting bit of code that explains the overall architecture of implementing a property using ScalaCheck. Detailed examples follow in upcoming sections and in the online code repository for the book.

The entire code fragment shows how to define a property. The text in ① indicates the predicate that the property is supposed to verify. Regardless of the type of accounts you deal with, a credit followed by a debit of equal amount must maintain the initial position of the account.⁶ You can also think of this statement as describing the property of encoding business rules (which is also true).

`forAll` is a function that creates a property from one or more *generators* and a *predicate*. A generator generates data that you use to verify your properties. A generator is modeled as `Gen[A]` in ScalaCheck and is the base contract for the abstraction that generates an instance of type `A`. In the simplest form, a `forAll` takes a `Gen[A]` and a predicate `A => Boolean` and checks for the satisfiability of the predicate. ScalaCheck comes with a suite of generators for the built-in types. You can also implement your own generators and supply them along with `forAll` to create custom properties relevant to your domain model. In the preceding example, the property definition takes two generators, `accountGen` and `moneyGen`, that generate instances of `Account` and `Money`, the two abstractions from our domain. These generators generate data that you use within the predicate to implement the domain rule that you need to verify. And this demonstrates one other feature of property-based testing: Instead of hardcoded data, you can pass generators that generate data for testing. Just imagine how clean your test suite would be if you could replace all hand-coded data with a few generators. You'll take a deeper look at generators in the next section and define a few of your own for defining properties of our model.

Properties and generators are the two most fundamental concepts of property-based testing. ScalaCheck offers lots of APIs for both of them that allow you to compose

⁵ And you must have seen all of these functions that I defined earlier in the book.

⁶ By position, I mean balance. And for simplicity, we assume a single-threaded model, and nothing happens between the credit and debit.

and build properties incrementally, thereby helping in the clean and noninvasive verifiability of much of your domain model.

9.5.2 Verifying properties from our domain model

Let's take an example of an abstraction from our domain model and step through some properties that you can define. Note that each of the following properties is a business rule that needs to be honored by our model implementation. You'll start with an implementation of `Account`, which you saw in earlier chapters. Chapter 3 presented a similar model when I talked about smart constructors. I also used this exact same model in chapter 6 when I discussed reactive domain models. The online repository of the book contains this implementation as part of chapter 6 as well as the current chapter. This text highlights the relevant portions of the model for brevity; you can look at the details of the implementation in the repository. The following listing contains the `Account` abstraction along with the algebraic data types for `CheckingAccount` and `SavingsAccount`.

Listing 9.3 The Account abstraction

```
import java.util.{ Date, Calendar }
```

```
object common {
    type Amount = BigDecimal
    def today = Calendar.getInstance.getTime
}
```

```
import common._

case class Balance(amount: Amount = 0)
```

```
sealed trait Account {                                     ←
    def no: String
    def name: String
    def dateOfOpen: Option[Date]
    def dateOfClose: Option[Date]
    def balance: Balance
}
```

```
final case class CheckingAccount (no: String, name: String,
    dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
    balance: Balance = Balance()) extends Account           ←

```

```
final case class SavingsAccount (no: String, name: String,
    rateOfInterest: Amount,
    dateOfOpen: Option[Date], dateOfClose: Option[Date] = None,
    balance: Balance = Balance()) extends Account           ←
```

Then you define some smart constructors in the following listing that allow you to create valid instances of `CheckingAccount` and `SavingsAccount`. Chapter 3 covered smart constructors; they provide access points for creating valid instances of domain objects.

Listing 9.4 Smart constructors for Account

```
object Account {
  def checkingAccount(no: String, name: String, openDate: Option[Date],
                      closeDate: Option[Date],
                      balance: Balance): NonEmptyList[AccountException] \ / Account = {
    val od = openDate.getOrElse(today)
    (
      validateAccountNo(no) |@|
      validateOpenCloseDate(openDate.getOrElse(today), closeDate)
    ) { (n, d) =>
      CheckingAccount(n, name, d._1, d._2, balance)
    }.disjunction
  }

  def savingsAccount(no: String, name: String, rate: BigDecimal,
                     openDate: Option[Date], closeDate: Option[Date],
                     balance: Balance): NonEmptyList[AccountException] \ / Account = {
    val od = openDate.getOrElse(today)
    (
      validateAccountNo(no) |@|
      validateOpenCloseDate(openDate.getOrElse(today), closeDate) |@|
      validateRate(rate)
    ) { (n, d, r) =>
      SavingsAccount(n, name, r, d._1, d._2, balance)
    }.disjunction
  }
}
```

Smart constructors ensure that you get a valid instance on creation.

Smart constructors ensure that you get a valid instance on creation.

This code is self-explanatory, especially because by now you must be familiar with all the idioms that the smart constructors use here. A fair number of validations within each of the smart constructors ensure that you get a valid instance of `CheckingAccount` or `SavingsAccount` upon successful construction. The validations are stitched together using applicative builders so that all validation errors accumulate together before you get them in the return type of the function. And, finally, listing 9.5 presents some domain functionalities that an `Account` needs to implement to perform the desired behaviors in the model. These provide a basis for defining properties for verification. For brevity, this listing elides over all the validation functions; check out the online repository for each of the implementations.

Listing 9.5 Domain functionalities on Account

```
object Account {
  def close(a: Account, closeDate: Date)
  : NonEmptyList[AccountException] \ / Account = {
  (validateAccountAlreadyClosed(a) |@|
  validateCloseDate(a, closeDate)) { (acc, d) =>
    acc match {
      case c: CheckingAccount => c.copy(dateOfClose = Some(closeDate))
    }
  }
}
```

```

        case s: SavingsAccount  => s.copy(dateOfClose = Some(closeDate))
    }
}.disjunction
}

def updateBalance(a: Account, amount: Amount)
: NonEmptyList[AccountException] \ / Account = {
(validateAccountAlreadyClosed(a) |@|
checkBalance(a, amount)) { (_, _) =>
a match {
case c: CheckingAccount =>
c.copy(balance = Balance(c.balance.amount + amount))
case s: SavingsAccount  =>
s.copy(balance = Balance(s.balance.amount + amount))
}
}.disjunction
}
}
}

```

VERIFYING ACCOUNT CLOSE BEHAVIOR

Consider a business rule that says, *If you have a customer account that hasn't been already closed, you should be able to close the account with a valid close date.* Let's try to write a property that verifies this business rule for every set of valid customer accounts and valid close dates for those accounts.

Listing 9.6 Verifying account close behavior algebraically

```

property("Close Account if not already closed") =
forAll(validCheckingAccountGen) {
_.map { account =>
account.dateOfClose.map(_ => true).getOrElse(
close(account,
account.dateOfOpen.map(aDateAfter(_)).getOrElse(common.today)
).isRight == true
)
}.getOrElse(false)
}

```

The properties that `forAll` generates are *universally quantified*. Given a generator, `forAll` creates properties for *every* value that the generator can generate. In listing 9.6, properties will be created for every account that the generator `validCheckingAccountGen` generates. You don't have to pass in handcrafted values of `Account` for creation of properties. The generator `Gen[A]` that forms the base of every generator that you implement is parametric on type `A`, so all properties are universally quantified. Note the predicate within `forAll` in listing 9.6. It takes an account that the generator generates and returns `true` if the account is already closed. Otherwise, it calls the `close` function with a valid close date and returns `true` if `close` is successful.

The main takeaway from this example is to appreciate that you've been able to verify a domain behavior in a completely declarative way using algebraic reasoning. The

property that listing 9.6 implements is equivalent to the algebraic expression *for all accounts (a) generated by some generator, close-account-if-not-already-closed must be successful.*

VERIFYING FAILURES WITH UPDATE BALANCE

Before you explore defining generators, let's look at yet another business rule verification through algebraic reasoning using properties. Listing 9.5 contains a domain function, updateBalance, that updates the balance of an account. This update can be in the form of a debit or withdrawal of funds, or a credit or deposit of funds. If you try to do a debit from an account that doesn't have a sufficient balance, the transaction must fail. Also if you try to do the same operation on an already closed account, that operation should also fail. Both validations are performed in the function updateBalance in listing 9.5. Let's verify that the validation works correctly in the following listing.

Listing 9.7 Verifying update balance behavior algebraically

```
property("Update balance on closed account fails") =
  forAll(validClosedCheckingAccountGen, genAmount) { (creation, amount) =>
    creation.map { account =>
      updateBalance(account, amount) match {
        case -\/(NonEmptyList(AlreadyClosed(_))) => true
        case _ => false
      }
    }.getOrElse(false)
  }

property("Update balance on account with insufficient funds fails") =
  forAll(validZeroBalanceCheckingAccountGen, genAmount) {
    (creation, amount) =>
    creation.map { account =>
      updateBalance(account, -amount) match {
        case -\/(NonEmptyList(InsufficientBalance(_))) => true
        case _ => false
      }
    }.getOrElse(false)
  }
```

Here, you use generators that generate the *appropriate type of data*. In the first example, the generator generates closed accounts. If you're wondering how it does that, you'll see in the next section. And in the second example of listing 9.7, the generator generates accounts with zero balance. In the first case, you declare the property verified when updateBalance fails with an exception `AlreadyClosed`. And in the second case, it fails with exception `InsufficientBalance`, as per the correct business rule. For brevity, I don't provide the implementation details of the exceptions here; you can get them from the online code repository for the book.

The important takeaway from this example is similar to the one we discussed for closed accounts—verifying business rules through algebraic properties. In this way, you can encode business rules algebraically for the whole of your model and make your model verifiable and robust.

Verification through types or algebraic properties?

One common theme throughout this book is the increased importance of using types for modeling. When you have a language such as Scala that offers a strong type system, why not take advantage of it? Types provide a great way to encode domain logic, and you've seen that many times in this book. When you encode logic with types, you don't have to write explicit tests; the compiler does that automatically.

Then why do you need algebraic properties to verify domain behaviors?

Types can't express every domain constraint or rule. You can't express the rules that we discussed in the context of listings 9.6 and 9.7 only with types. You need explicit tests for their verification. The beauty of property-based testing is that it allows you to generate universally quantified properties and hence makes your testing process much more exhaustive than you can have with manually crafted datasets.

The core point is that you need both. Use types wherever you can. Otherwise, encode properties algebraically to verify business rules.

9.5.3 Data generators

Besides properties, generators are the second leg that gives power to the technique of property-based testing. Universally quantified properties need data generators for exhaustive testing. The combination of the two makes the process of verification more complete and scalable than xUnit-based testing with manually crafted data.

In ScalaCheck, the generator comes from the `org.scalacheck.Gen` class. You can think of `Gen[A]` as a function from `Gen.Params => Option[A]`. But the class `Gen` offers enough combinators that provide lots of helpful ways to generate data. You can go through the entire list of combinators from the documentation of ScalaCheck at www.scalacheck.org. But to get an idea of some of the capabilities of `Gen` and how you can combine them using for-comprehensions to generate more complex data, here's a sample session through the Scala Repl:

```
scala> import org.scalacheck._  
import org.scalacheck._  
  
scala> import Prop.forAll  
import Prop.forAll  
  
// generate an integer between 10 and 20  
scala> Gen.choose(10, 20)  
res0: org.scalacheck.Gen[Int] = org.scalacheck.Gen$anon$3@3bb613d7  
  
// generate samples from the above generators  
scala> res0.sample  
res1: Option[Int] = Some(11)  
  
scala> res0.sample  
res2: Option[Int] = Some(17)  
  
scala> res0.sample  
res3: Option[Int] = Some(16)
```

```
// compose generators using for comprehension
scala> for {
|   a <- Gen.choose(10, 20)
|   b <- Gen.choose(100, 200)
| } yield (a, b)
res4: org.scalacheck.Gen[(Int, Int)] = org.scalacheck.Gen$anon$6@77146bb6

scala> res4.sample
res5: Option[(Int, Int)] = Some((19,119))

scala> val vowel = Gen.oneOf('A', 'E', 'I', 'O', 'U', 'Y')
vowel: org.scalacheck.Gen[Char] = org.scalacheck.Gen$anon$3@6f8751cb

scala> vowel.sample
res6: Option[Char] = Some(O)

scala> vowel.sample
res7: Option[Char] = Some(U)

scala> case class Balance(amount: BigDecimal)
defined class Balance

scala> val genAmount = for {
|   value <- Gen.chooseNum(100, 10000000)
|   valueDecimal = BigDecimal.valueOf(value)
| } yield valueDecimal / 100
genAmount: org.scalacheck.Gen[scala.math.BigDecimal] = ...

scala> val genBalance = genAmount map Balance
genBalance: org.scalacheck.Gen[Balance] = ...

scala> genBalance.sample
res8: Option[Balance] = Some(Balance(1))

scala> genBalance.sample
res9: Option[Balance] = Some(Balance(76918.86))

scala> genBalance.sample
res13: Option[Balance] = Some(Balance(14665.5))
```

With an idea of how generators work in ScalaCheck, here's one from our example in listing 9.6. `validCheckingAccountGen` generates valid `CheckingAccount` instances so that you can create the properties that verify account close behavior on them. The following listing implements `validCheckingAccountGen`.

Listing 9.8 Sample data generator for CheckingAccount

```
import java.util.Date
import org.scalacheck._
import Prop.forAll
import Gen._
import Arbitrary.arbitrary
```

```

val amountGen = for {
    value <- Gen.chooseNum(100, 10000000)
    valueDecimal = BigDecimal.valueOf(value)
} yield valueDecimal / 100

```

Generates a Balance out of the Amount

Generates an Amount, which is a BigDecimal

```

val balanceGen = amountGen map Balance

```



```

implicit val arbitraryBalance: Arbitrary[Balance] = Arbitrary(balanceGen)

```

Generates a valid account number

```

val validAccountNoGen = Gen.choose(100000, 999999).map(_.toString)

```

Generates a valid name

```

val nameGen = Gen.oneOf("john", "david", "mary")

```

```

def optionalValidCloseDateGen(seed: Date) =
  Gen.frequency(
    (8, Some(aDateAfter(seed))),
    (1, None)
  )

```

Generates a valid account close date. See text for details.

This allows using the “arbitrary” combinator for Balance generation.

①

```

val validCheckingAccountGen = for {
    no <- validAccountNoGen
    nm <- nameGen
    od <- arbitrary[Date]
    cd <- optionalValidCloseDateGen(od)
    bl <- arbitrary[Balance]
} yield checkingAccount(no, nm, Some(od), cd, bl)

```

②

③

Generates a checking account through the smart constructor

Although some of the combinators aren’t yet familiar to you, I’m sure you can get the general idea of how checking account generators work from this listing. Let’s look at some of the special features used to implement a generator for a nontrivial domain class like `CheckingAccount`:

- *Generating balance*—`amountGen` and `balanceGen` are straightforward and use the standard combinators of `Gen` to generate a `BigDecimal` and then map on to form a `Balance`. The only new thing is the `Arbitrary` abstraction in ① that enables you to use the `arbitrary` combinator in generating a `Balance`, which you use in `validCheckingAccountGen` ②.
- *Generating account number*—Note how you use the `choose` combinator to generate an integer in the specific range so that you can make it into a valid account number. The account number has to be of at least length 5 to be valid.
- *Generating account close dates*—This ③ uses some interesting features of data generation in ScalaCheck. You’re generating either a value of `None` or a close date that’s *after* the passed seed date (usually you’ll pass the account open date as seed). But you can control the frequency of generated data. Here you specify that you’d like to generate a valid close date with a frequency of 8 relative to `None`. And ScalaCheck generates data based on this distribution.

- *Generating valid checking accounts*—This is nothing but a monadic composition of the preceding generators. After you get all the data required to generate an account, you use the smart constructor `checkingAccount` to construct a valid instance. The smart constructor returns a disjunction (`\/`), so you need to apply appropriate transformations and verify properties for the generated account. You may want to go back and revisit listings 9.6 and 9.7 at this point and be sure that you understand the whole account-creation process in the context of property verification.

9.5.4 Better than xUnit-based testing?

As you saw in the previous sections, properties force you to think in terms of algebraic behaviors of the model. Compare that to xUnit-based testing, where you test a scenario with a few instances of handcrafted data. Which one do you think is more useful when it comes to testing complex domain models? Here are a few points that highlight some of the pros and cons of these approaches:

- *Intuitive*—An example-based approach with scenarios tested using handcrafted data may seem more intuitive to someone who isn't familiar with the domain. You see concrete data elements passed to your function or class and have assertions tested against them. This helps you correspond better with the underlying implementation. xUnit-based testing with concrete examples wins here.
- *Thinking in properties*—Property-based testing forces you to think harder in terms of the algebraic behavior of the model. This often leads to better coverage of test paths, because most properties tend to be cross-cutting in nature and map closely to the business rules of the domain. This exercise also maps more closely to the philosophy of functional programming, where you're encouraged to think algebraically.
- *Domain rule repository*—A rich set of properties serves as a blueprint of domain rules—an invaluable repository that leads to better verifiability of your domain model.
- *Data generation*—In an example-based approach with xUnit, you write handcrafted data and test the scenarios against the few that you can write manually. On the other hand, with property-based testing, you can generate lots and lots of data and hammer each of your functions against the entire set. You can control the volume and distribution of this data declaratively and get far better coverage of edge cases and boundary conditions in your testing. This is possibly the biggest advantage that you get with property-based testing.

9.6 Summary

This chapter covered how to design testable domain models and then use property-based testing to verify algebraic properties of your model. Here are the main points that were covered:

- *Testability is closely related to modularity.* You need to ensure that you have proper separation of concerns between your domain artifacts.
- *xUnit and its limitations.* You also saw the drawbacks of xUnit-based testing and how to use property-based techniques as an alternative.
- *Property based testing and its advantages.* Property-based testing allows you to generate data through generators, and you have fine-grained control over what data to generate. And then you can use your generators to define and verify algebraic properties for your domain model.

10

Summary—core thoughts and principles

This chapter covers

- The core guiding principles covered throughout this book
- A brief overview of some trends in domain modeling

This chapter presents a look back at some of the fundamental thoughts, principles, and idioms covered throughout this book. In the earlier chapters, I discussed lots of techniques for building domain models that are responsive and can be reasoned about out of pure functions in the domain of personal and investment banking. Some of those techniques are core principles, in the sense that you have to abide by them when you do your modeling. Others, which are more advanced, will make your code more elegant, efficient, and modular. We look back at some of them in the following sections.

10.1 Looking back

Now that you've been through the entire journey of how to think of domain models in terms of pure and compositional structures, let's review the major themes that form the core of our entire implementation thought process. These are the

ones that you should revisit every time you start implementing a domain model. These are not only implementation artifacts; many of them form the fundamental patterns that need to be part of your thought process when doing functional domain modeling. I've repeatedly mentioned that the functional approach is similar to the way we treat functions in mathematics—so if you don't think right, you won't end up with the ideal solution to your problem. And in many cases, you'll need iterations. Even after so many years of experience, mature designers rely on iterations and feedback before finalizing a specific model of their expected precision.

The first thing that you need to remember is that a domain model belongs to the problem domain. You need to be aware of the *model elements*, the *domain vocabulary*, and the *boundaries of the subsystems* before you start implementing the solution model. The implementation has to reflect all the behaviors that the problem domain has, with similar clarity and using the same ubiquitous language (the domain vocabulary). And as mentioned in many sections of the book, the solution has to present to the user an interface that is uncluttered, clean, domain friendly, and devoid of any incidental complexities of the underlying implementation. We discussed techniques for achieving these qualities by using various functional design patterns and reactive principles. Designing at the right level of abstraction is the key, and you'll revisit some of these patterns and principles in this chapter as a gentle reminder. You can always tinker with the details of implementation, but you need to stick to the core principles. Let's recap the more important ones next.

10.2 Rehashing core principles for functional domain modeling

In this section, you'll look back at some of the core design patterns from the previous chapters. The important takeaway from this discussion is a rehash of why these principles make sense when we use functional programming for domain modeling. I won't go into the benefits of functional programming, which I'm sure you're quite familiar with by this time. But to get the maximum leverage out of designing your domain model around pure functions, you need to honor some of the basic principles.

10.2.1 Think in expressions

An *expression* is something that has a value. On the other hand, statements yield a side effect. The value that an expression generates can be passed around in functions, generating bigger expressions. Evolve your domain behaviors as expressions generated bottom up from smaller ones so that you can compose them better with other behaviors. Let's recap an example from section 4.2.3 on monadic effects:

```
object App extends AccountService {
    def op(no: String) = for {
        _ <- credit(no, BigDecimal(100))
        _ <- credit(no, BigDecimal(300))
    } yield ...
```

```

    _ <- debit(no, BigDecimal(160))
    b <- balance(no)
  } yield b
}

```

The function `op` generates a value by evaluating the for-expression. Each clause within the for-comprehension is again an expression that generates separate values that get chained together through a sequence of `flatMap`s and `map`s to yield the final value. Thinking in expressions helps you think in terms of composition, making evolution of domain behaviors more natural and incremental. Instead of committing to the side effect early on, use combinators to abstract the effect so that you can compose into more complex behaviors without breaking the chain of expressions.¹

10.2.2 Abstract early, evaluate late

In section 3.2.1, you saw how to differentiate between computations and values. A *computation* is an abstraction over values; for example, `Try` abstracts over failures. While implementing domain models, don't commit to values early on; you'll lose the power to compose the specific behavior with other computations. If you have a function, `debit`, that can fail in the face of insufficient account balance, return a `Try` or `scalaz._` as a computation that abstracts the outcome of the operation. Alternatively, you could return a value indicating a success or failure. But in doing that, you can't sequence the `debit` operation with other operations downstream. For example, you couldn't compose them into a for-comprehension just as you did in the preceding section. Commit to a value only when you've reached the end of the sequence of computations.

10.2.3 Use the least powerful abstraction that fits

Initially, this may seem counterintuitive. But consider that the more powerful an abstraction is, the more specialized it is. And so you can use it in less generalized circumstances. If you use an abstraction that's more powerful than what you need, you lose out on the reusability part. We discussed this at length in section 4.2.3 when talking about differences between monadic and applicative effects. Monads are more powerful and hence less reusable than applicatives. And we concluded that you should use applicatives whenever you can; your abstraction becomes usable in more contexts. I mention this principle in the context of functional programming, but this is a hugely useful principle to follow in any paradigm of modeling or design.

¹ Just as you did in section 3.2 with `Try` or disjunction to abstract failures.

10.2.4 Publish what to do, hide how to do within combinators

One of the core strengths of functional programming is that it provides you the right tool to declare *what* you want to do instead of *how* to do it. The *how* part can be abstracted within functions that you can compose together with the *what* part of your implementation. Consider the following example from listing 4.10 in section 4.4.3. The implementation of `tradeGeneration` looks like a copy from the business specification document that mentions *what* steps you need to follow to generate trades. How the various steps combine together both in the happy path and in the exceptional path of execution is abstracted within the combinator `andThen`. If you followed the example in section 4.4.3, you must have noticed that `andThen` is a higher-order function defined in the `Kleisli` class and uses the algebra of a monad to provide the desired effect within your API. All this power comes from the ability to compose functions and reuse existing algebra and helps make your implementation succinct yet expressive.²

```
def tradeGeneration(
    market: Market,
    broker: Account,
    clientAccounts: List[Account]
) = {
    clientOrders andThen
        execute(market, broker) andThen
            allocate(clientAccounts)
}
```

10.2.5 Decouple algebra from the implementation

This is one of the most fundamental principles of system design. Interfaces, or contracts, are published to the client and hence can't be varied at will. But underlying implementations can change and hence need to be decoupled from the published contract. This book has focused on algebra-based design; the algebra that we refer to is the *interface*, the *contract* of your API with the client. For one algebra, you can have multiple implementations, but you need to be careful so as to provide the correct level of isolation between the two. Chapter 5 covered many ways of achieving this goal:

- Using traits-based modular composition that terminates into concrete implementation of objects (section 5.2)
- The Type Class pattern (section 5.3)
- Free monads and the interpreter pattern (section 5.5)

Modularization is one of the primary things that makes your domain model composable and manageable in the face of changes. Use any or all of these patterns as per your requirements.

² Here `andThen` reuses the algebra of a monad.

10.2.6 Isolate bounded contexts

Modules lead naturally to bounded contexts. A *bounded context* is an independent domain model with its own ubiquitous language, entities, and types. Any nontrivial domain model has multiple models within it, each modeling a separate bounded context. Identifying a bounded context is possibly the first iterative exercise that you should do as part of domain modeling. We discussed bounded contexts in detail in chapter 5, and you saw various ways to communicate across them. In many situations, you'll realize that individual bounded contexts often can evolve to microservices—units of the whole model that have separate data-type semantics, entities, and relationships, and a domain vocabulary.

10.2.7 Prefer futures to actors

In the community today, actors are marketed with more velocity than futures. But you must remember that futures compose and hence make a better substrate for building larger behaviors from smaller ones. I discuss this in detail in chapter 6, which describes the design principles of reactive systems. When looking to build nontrivial asynchronous abstractions, always reach for futures first.³ This doesn't mean that actors don't have any role to play in your domain model. Chapter 6 provides specifics about some of the meaningful use cases where actors make sense. After all, it's always good to have more tricks up your sleeve when you embark upon the job of implementing a complex domain model.

10.3 Looking forward

As its central theme, this book has emphasized the power of functions and their compositionality in building generic abstractions that form the core substrate of domain models. Instead of focusing on objects, you need to start thinking of a model in terms of behaviors. The reason is simple: Behaviors (functions) can compose mathematically. And the moment you map your model elements to the world of mathematics, you open up the world of mathematically verified models. Scala isn't that pure of a language. But languages such as Haskell are much purer, and functions in Haskell are much like equations in mathematics. You can do *equational reasoning* on your functions and prove that they behave in the way they're supposed to. The primary reason Haskell makes equational reasoning much more feasible than Scala is that the Haskell type system supports a much stricter separation of side effects from pure logic. The type system in Haskell will ensure that the functions you're manipulating as mathematical equations do indeed honor the algebra and don't sneak in other unintended side effects. You can get close with Scala too, if you can impose certain discipline in your programming idioms. But with Haskell, the reasoning part is much more sound than with Scala. Modeling your domain with a language like Haskell is definitely one of the future trends that I look forward to.

³ Also have a look at `scalaz.concurrent.Task`, which is discussed in chapter 6.

Another area that's being developed a lot these days and could add much value in developing statically verified domain models is *dependent typing*. Many of the domain constraints or rules that we encode using procedural logic can be encoded as part of the type system itself. Idris (www.idris-lang.org) is an example of a dependently typed language that offers these capabilities. Haskell (https://wiki.haskell.org/Dependent_type) has also started developing dependent typing features. And in Scala, you use Shapeless (<https://github.com/milessabin/shapeless>) to do a lot of these things, though in a slightly more verbose way. More power to the type system means more succinct encoding of domain logic, which in turn leads to fewer and fewer tests needing to be written.

As models become more and more complex, we, as modelers, are facing lots of pressure to make those models more responsive. You've seen how the word *reactive* has become more mainstream these days. We have the reactive manifesto and we're seeing the release of more libraries and frameworks that claim to be reactive. In chapters 6 and 7, you learned a lot about how to make models reactive by using the various tools for concurrency and parallelism that Scala offers. Specifically, we discussed Akka (<http://akka.io>), which has become one of the most popular toolkits on the JVM for developing reactive applications.

One of the trends in reactive application development that's fast gaining steam is the use of streaming capabilities in delivering domain services. You've seen Akka Streams, one of the popular streaming libraries that comes with Akka. You may be wondering why streams are becoming so popular in the context of domain modeling. The reason is this necessity to provide more complex behaviors involving lots of data flow as part of the core business in a nonblocking way for increased responsiveness in domain services. Just the other day, we were doing these chores as map-reduce-powered batch processes within a separate platform for analytics. The need to deliver real-time solutions is rapidly increasing, and stream processing is one of the major techniques to get there. Already we've been seeing libraries such as Apache Spark (<http://spark.apache.org>), Apache Flink (<http://flink.apache.org>), Apache Samza (<http://samza.apache.org>), and Apache Storm (<http://storm.apache.org>) as some of the major players in this league. And they're increasing by the day. Your domain model needs to handle streaming and integrate with these libraries. Ironically, the challenge is to abide by the age-old concern of software engineering—*separation of concerns*. You need to think about how to keep core domain behaviors decoupled from technology concerns such as streaming or batching. And again, you'll find that the principles of modularization that functional programming offers will come to your rescue. Streaming platforms such as Spark or Flink use functional programming principles, with Scala as the implementation language. This isn't without a reason; finally, we've found that the mathematical foundations of functional programming provide the strongest basis to modularize our domain models.

index

Symbols

\vee symbol 158, 170, 185
 \rightsquigarrow symbol 177

A

abstract functions 123
abstract syntax tree. *See* AST
abstract types 65
abstracting operations 114
abstractions
 composing 77–79
 early 281
 least powerful 281
 over evaluation 76–77
 overview 121
 purity of 53–55
Account abstraction 57, 270
Account aggregate 11–13, 20, 245
Account class 16, 164
Account domain model 191
account repository 169–175
 composing DSL 174–175
 defining building blocks 173
 defining module 173–174
Account Service module 151
accountGen generator 269
account-identifying attributes 10
AccountRepo monad 173
AccountRepoInterpreter 193
AccountRepository 13, 156, 184, 203, 265
AccountService module 79, 116, 127, 184, 195
accumulate function 266
ACID (Atomic, Consistent, Isolated, and Durable) 232

actor model 205–211
 bounded context 209–210
 composability 207
 functional domain models 210–211
 overview 205–206
 reactive API design 207–209
 resilient models 210
 type safety 207
actor-based implementation 208
ActorMaterializer 200, 211, 219
actors, vs. futures 283
ActorSubscriber 218
ad hoc polymorphism 164, 166
addressNoLens 95
ADTs (algebraic data types) 56–61
 immutability encouraged by 60–61
 overview 19
 pattern matching and 59–60
 structure data in model 58
 sum type and product type 56–57
aggregate root 11
aggregate transactions 216
aggregates 10–13
 updating functionally with lenses 92–97
 with algebraic data types 89–92
 invariants 91–92
 modularity 91
Akka Streams 258–259
algebra
 of API design 74–75
 decoupling from implementation 282
 functional patterns and 140–143
algebra for domain service 76–83
 abstractions
 composing 77–79
 over evaluation 76–77

algebra for domain service (*continued*)
 final algebra of types 79–80
 interpreter for algebra 82–83
 laws of algebra 81
 algebraic data types. *See* ADTs
 algebraic properties 262, 266, 268, 273–274, 278
 allocate 189
 allocation 140
 amountGen 276
 Analytics module 112
 andThen 282
 anticorruption layer 168
 antipattern 17
 API design, algebra of 74–75
 app package 161
 Application box 156
 applicative effects, difference from monadic effects 127–129
 Applicative Functor pattern 118, 122–125, 136–137
 apply method 248, 250
 apply3 121
 AST (abstract syntax tree) 171, 178
 asynchronous boundaries 196–197
 asynchronous messaging, explicit 196–197
 asynchrony, as stackable effect 185–187
 monad transformers 185–186
 stacking Future and Either 186–187
 Atomic, Consistent, Isolated, and Durable. *See* ACID
 auditing 237
 automatic failure handling 129

B

back-end query power 258
 back-pressure handling 198, 204–205
 Balance type 16, 256
 BalanceComputation 63
 balanceGen 276
 BankingService 7
 become method 206–207
 big data 205
 BigDecimal 68–69, 111
 binary operation 111, 113, 134
 bind method 80
 boundaries of subsystems 280
 bounded contexts 166–168
 actors and 209–210
 communication between 168
 isolating 283
 modules and 167–168
 bounded latency 182, 198
 building abstractions 101
 building blocks 171

C

case classes 12, 20, 60
 centralized handlers 34
 channels 215
 close command 246
 close operation 81
 closeAccount function 118
 clustering 58, 226
 code modularization 42
 collaboration 231
 combinators 24–26, 282
 command handlers 240, 250
 command processor 241
 Command Query Responsibility Segregation pattern. *See* CQRS pattern
 Command trait 246
 command-processing loop 246
 commands 38–39, 235, 245–247
 communication pathways 168
 complexities 3
 composability, actors and 207
 composable modules 150
 compose function 95
 composed DSL 172
 composite data type 247
 composition function 22–27
 composition of futures 71
 compositional domain models 29
 compositionality property 23, 75, 154, 163
 compositionality, lack of 99
 computation 77
 concrete implementations 162
 conformist context 197
 consistency boundary 10–11
 copy function 94, 96
 coupling API context 99
 CQRS (Command Query Responsibility Segregation) pattern 235–237
 distributed 253–254
 implementing event sourcing and 240–242
 batching commands 241–242
 handling side effects 242
 overview 211
 CRUD (Create, Retrieve, Update, and Delete) 232
 Currency data type 56
 curried argument 100–101

D

data generation 277
 data generators 274–277
 DayOfWeek 87

- DDD (domain-driven design) 4–15
 bounded context 5
 domain model elements 5–9
 lifecycle of domain object 9–14
 aggregates 10–13
 factories 9–10
 repositories 13–14
 ubiquitous language 14–15
 declarative APIs 223
 decoupling side effects 28
 denormalized data model 166
 denormalized view 234
 denotation 172
 dependencies, injecting 103
 dependency 158
 dependent typing 284
 design for failure 33, 67
 design pattern 109
 Disjunction type 123
 distributed actors 207
 domain elements 8
 domain events 39–41, 238–239
 domain logic 241
 domain models 73–107, 109–134, 139–179
 algebra 140–143
 algebra of API design 74–75
 Applicative Functor pattern 118–125
 bounded contexts 166–168
 communication between 168
 modules and 167–168
 case study 152–163
 anatomy of module 152–158
 composition of modules 159–160
 modularity encourages compositionality 162–163
 physical organization of modules 160–162
 defining algebra for domain service 76–83
 abstracting over evaluation 76–77
 composing abstractions 77–79
 final algebra of types 79–80
 interpreter for algebra 82–83
 laws of algebra 81
 elements of 5–9
 free monads 169–179
 account repository 169–170, 172–175
 interpreters for 175–177
 making it free 170–172
 functors 117–118
 lifecycle patterns 83–105
 aggregates, updating functionally with lenses 92–97
 aggregates, with algebraic data types 89–92
 effective use of 104–105
 expressive types 88–89
 factories 85–86
 repositories and decoupling 97–103
 smart constructor idiom 86–87
 mining in 110–111
 modularizing polymorphic behaviors 163–166
 monadic effects 125–134
 difference from applicative effect 127–129
 generic monad module 126
 monad combinators 132–134
 state monad 130–132
 parametric 111–116
 abstracting over context 114–116
 abstracting over operations 113
 identifying commonality 111–113
 reactive domain models 32–36
 3+1 view of 33
 debunking 33–35
 elasticity and 35–36
 shaping 134–138
 tightening up domain invariants with 144–147
 type class pattern 163–166
 types 143–144
See also functional domain model design; testing domain model
 domain object, lifecycle of 9–14
 aggregates 10–13
 factories 9–10
 repositories 13–14
 domain rules 74
 domain services 161
 domain validations 250
 domain vocabulary 280
 domain-driven design. *See* DDD
 domain-model elements 216
 DomainShowProtocol module 165
 domain-specific language 15
 DSL, composing 174–175

E

-
- edges 214
 effectful computation 67, 108, 118
 Either data type 56–57, 76, 158, 186–187
 elastic APIs 205
 elastic attribute 32
 end of the world 163
 entities 5
 equational reasoning 29, 31–32, 283
 Erlang 205, 224
 essential complexities 3
 evaluation, substitution model of 30
 Event abstraction 243

event sourcing 226, 237–242
 commands and events in event-sourced domain
 model 238–240
 command handlers 240
 domain events 238–239
 implementing CQRS and 240–242
 batching commands 241–242
 handling side effects 242
 implementing event-sourced domain
 model 242–255
 commands as free monads over events 245–247
 distributed CQRS 253–254
 event store 253
 events as first-class entities 243–245
 interpreters 247–252
 projections—read side model 252
 using free monads 243
 event store 241, 249, 253
 Event type 246
 event-driven programming 38–41
 domain events 39–41
 events and commands 38–39
 events
 as first-class entities 243–245
 overview 38–39
 event-sourced domain model, commands and
 events in 238–240
 command handlers 240
 domain events 238–239
 Eventuate 254
 exception-handling logic 34
 exceptions, in Scala 22
 explicit asynchronous messaging 196–197
 explicit collaborations 154
 expression-oriented programming 51
 expressions, thinking in 280–281
 expressive types 88–89
 external sources 34
 extractors 105

F

factories 9–10, 85–86
 fail-fast validation 138
 Failure construct 20, 159
 failure handling 129
 Failure variant 69
 filter combinator 51
 final keyword 89
 financial instruments 190
 finite-state machine. *See* FSM
 flatMap method 26, 69–70
 FlexiMerge operation 203
 Flow abstraction 199
 flow of information 198

foldLeft 51
 foldMap function 248
 for loop 51
 forAll function 269
 foreach combinator 26
 Framing class 220
 free monads 169–179
 account repository 169–170, 172–175
 composing DSL 174–175
 defining building blocks 173
 defining module 173–174
 interpreters for 175–177
 making it free 170–172
 FSM (finite-state machine) 206–207
 function body 15
 function composition 22–27, 129, 132, 134,
 138–139, 142, 148
 functional abstractions 22, 123, 135, 137
 functional combinators 125, 257
 functional domain model design 73–106
 actors and 210–211
 algebra of API design 74–75
 defining algebra for domain service 76–83
 abstracting over evaluation 76–77
 composing abstractions 77–79
 final algebra of types 79–80
 interpreter for algebra 82–83
 laws of algebra 81
 lifecycle patterns 83–105
 aggregates, updating functionally with
 lenses 92–97
 aggregates, with algebraic data types 89–92
 effective use of 104–105
 expressive types 88–89
 factories 85–86
 repositories and decoupling 97–103
 smart constructor idiom 86–87
 functional patterns 107, 109–134, 139–148
 algebra 140–143
 Applicative Functor pattern 118–125
 functors 117–118
 mining in domain model 110–111
 monadic effects 125–134
 difference from applicative effect 127–129
 generic monad module 126
 monad combinators 132–134
 state monad 130–132
 shaping of domain model by 134–138
 tightening up domain invariants with 144–147
 types 143–144
 using to make domain models parametric
 111–116
 abstracting over context 114–116
 abstracting over operations 113
 identifying commonality 111–113

functors 117–118
 fusion 54
 Future, stacking 186–187
 Future-based computation 125
 futures, vs. actors 283

G

generateAuditLog function 22, 26
 generatePortfolio 38
 Generator class 133
 generic monad module 126
 generically composing code 137
 geocoding service 9
 getEquityPortfolio method 192
 getHolding function 60
 getter 93
 graph.run() method 204
 graphs, as domain pipeline 202–204
 groupBy 201

H

handcoded data 269
 handleCommand function 250
 handlers 34
 handling exceptions 20
 happy path 68
 Haskell 284
 higher-order functions 24–25

I

identity verification 29
 Idris 284
 immutability of entities 7
 immutability, ADTs and 60–61
 immutable abstraction 12
 immutable characteristic 239
 implementation independence 154
 incidental complexities 3
 incremental composition 101
 infinite data 205
 injecting dependencies 103
 input 18
 inside-out effect 183
 instantiating accounts, in Scala 10
 Instrument domain model 191
 interest computation 65
 InterestBearingAccount 48
 InterestCalculation 65, 160
 interpreter package 161
 interpreters 82–83, 171, 177, 247–252

invariants
 overview 74, 91–92
 tightening up with functional patterns 144–147
 I/O exception 27
 isValid function 219

J

java.io.Serializable 163
 JUnit 266
 JVM (Java virtual machine) 44

K

Kleisli arrow 142
 Kleisli class 158, 184, 282
 Kleisli composition 129, 142
 Kleisli pattern 142–143, 145, 147

L

ladder of goodness 135
 latency, reducing with parallel fetch 189–193
 financial instruments 190
 overview 32
 portfolio service 190
 lawful abstractions 85
 lens laws, verifying 97
 lenses 93, 96
 Let It Crash philosophy 210
 lifecycle of domain object 9–14
 aggregates 10–13
 factories 9–10
 repositories 13–14
 lifecycle patterns 83–105
 aggregates, updating functionally with lenses 92–97
 aggregates, with algebraic data types 89–92
 invariants 91–92
 modularity 91
 effective use of 104–105
 expressive types 88–89
 factories 85–86
 repositories and decoupling 97–103
 injecting repository into service 98–99
 making repository curried argument to service methods 100–101
 reader monad 101–103
 smart constructor idiom 86–87
 liftEvent function 246
 local execution model 37
 location transparency 206

M

manage failures 67
 manipulating data, functionally 257–258
 map combinator 50–51
 map expression 54
 map function 24, 117, 216
 mapAsync function 216
 mapping aggregates, as ADTs to relational tables 255–257
 mapReduce 266
 materializer 200
 MergePreferred operation 203
 message-driven attribute 32, 36
 messaging protocol 168
 microservices 254
 minimally valid 84–85
 mixins 12, 63
 model elements 160, 280
 model package 161
 model testing 260–278
 algebra of model 267
 designing testable domain models 262–266
 decoupling side effects 263–264
 implementing parametricity and testing 265–266
 providing custom interpreters for domain algebra 264–265
 overview 260–262
 property-based testing 268–277
 data generators 274–277
 modeling properties 268–270
 verifying properties from domain model 270–274
 vs. xUnit-based testing 277
 xUnit-based testing 266–267
 modeling currency 56
 modeling data 223
 modified state 17
 modify combinator 132
 modular system 4
 modularization of domain models 149–179
 bounded contexts 166–168
 communication between 168
 modules and 167–168
 case study 152–163
 anatomy of module 152–158
 composition of modules 159–160
 modularity encourages compositionality 162–163
 physical organization of modules 160–162
 free monads 169–179
 account repository 169–170, 172–175
 interpreters for 175–177
 making it free 170–172

modularizing polymorphic behaviors 163–166
 overview 150–151
 type class pattern 163–166
 module organization 163
 modules 62–66, 74, 84, 150
 monad for free 172
 Monad trait 129
 monad transformer-based implementation 187–189
 handling effects algebraically 188–189
 reasons for using 188
 monad transformers 185–186
 monadic effects 125, 128–134
 difference from applicative effect 127–129
 generic monad module 126
 monad combinators 132–134
 state monad 130–132
 monadic model 77
 monads 80
 monoid 110
 morphisms 116
 multiple dependencies, injecting 103
 mutable state 16–19, 24, 29, 35, 42
 mutating objects 12
 mutation, of shared state 237

N

natural transformation 177
 nonblocking API design with futures 184–195
 asynchrony as stackable effect 185–187
 monad transformers 185–186
 stacking Future and Either 186–187
 monad transformer-based
 implementation 187–189
 handling effects algebraically 188–189
 reasons for using 188
 reducing latency with parallel fetch 189–193
 financial instruments 190
 portfolio service 190
 using scalaz.concurrent.Task as reactive construct 193–195
 nonblocking APIs 258
 nonblocking communication 35
 nonblocking elements 47
 noncompositional domain models 29
 Nondeterminism class 193
 nonidentifying attributes 6, 10
 NoSQL stores 253
 NUnit 266

O

OO (object-oriented) 15, 44
 operational complexity 255

operations 74
 operators 215
 Option data type 76
 organizing repositories, in modules 98
`org.scalacheck.Gen` class 274
`OutOfMemoryError` 204
 outside-in effect 183

P

pairwise elements 111
 parallel execution 56
 parallel fetch, reducing latency with 189–193
 parameterized types 154
 parametricity 93, 110, 115, 135, 144
 participation phase 84
 pattern matching
 ADTs and 59–60
 overview 87, 91, 104
 patterns 9, 107, 109–134, 139–148
 algebra 140–143
 Applicative Functor pattern 118–125
 functors 117–118
 mining in domain model 110–111
 monadic effects 125–134
 difference from applicative effect 127–129
 generic monad module 126
 monad combinators 132–134
 state monad 130–132
 shaping of domain model by 134–138
 tightening up domain invariants with 144–147
 types 143–144
 using to make domain models parametric 111–116
 abstracting over context 114–116
 abstracting over operations 113
 identifying commonality 111–113
 persistence event 9
 persistence of data 226–228
 persistence of domain models
 overview 231–233
 separation of concerns 233–237
 CQRS pattern 235–237
 read and write models of persistence 234–235
 persistence stage 83
 persistent actors 225–226
`PersistentActor`, Summarizer as 227–228
 Phantom Type pattern 147
 phantom types 146–148
 polymorphic functions 47
 Portfolio ADT 91
 portfolio service 190
`PortfolioGeneration` 63
`PortfolioService` module 47, 192, 194
`postTransactions` function 162

private implementation 163
 private specifier 89
 processing commands 249
 product type 56–57, 60
 projections 252
 properties 29
 property-based testing 260, 268–269, 274, 277–278
 data generators 274–277
 modeling properties 268–270
 verifying properties from domain model 270–274
 account close behavior 272–273
 failures with update balance 273–274
 vs. xUnit-based testing 277
 protocol implementations 165
 protocol selection 166
 published contracts 163
 publish-subscribe characteristic 239
 publish-subscribe model 168
 pull model 252
 pure data type 171–173, 179, 243, 246–247
 pure functions for domain behavior
 benefits of being referentially transparent 55–56
 parallel execution 56
 testability 55
 overview 49–56
 purity of abstractions 53–55
 pure model elements, virtues of 29–32
 purified function 264
 push model 252

Q

queries 235

R

RDBMS (relational database management system) 13
 RDBMS-based CRUD model 232
 reactive API design, actors and 207–209
 reactive domain models 32–36, 180–212
 3+1 view of 33
 actor model 205–211
 bounded context 209–210
 composability 207
 functional domain models 210–211
 reactive API design 207–209
 resilient models 210
 type safety 207
 debunking 33–35
 elasticity and 35–36
 explicit asynchronous messaging 196–197

reactive domain models (*continued*)
 nonblocking API design with futures
 184–195
 asynchrony as stackable effect 185–187
 monad transformer-based
 implementation 187–189
 reducing latency with parallel fetch
 189–193
 using scalaz.concurrent.Task as reactive
 construct 193–195
 overview 181–184
 stream model 197–205
 back-pressure handling 204–205
 graph as domain pipeline 202–204
 sample use case 198–202
 reactive fetch, that pipelines to Akka Streams
 258–259
 Reactive Manifesto 228
 reactive models
 attributes of 32
 overview 45
 reactive streams model 213–229
 back office implementation 220–223
 Summarizer actor and 221–222
 transaction-processing stream pipeline
 222–223
 domain use case 216–217
 front office implementation 218–219
 making models resilient 224–228
 clustering for redundancy 226
 persistence of data 226–228
 supervision with Akka Streams 225
 overview 214–215
 reactive principles 228–229
 stream-based domain interaction 217–218
 when to use 215–216
 read and write models of persistence 234–235
 read model 236
 reader monad 101–103
 ReaderT 101
 Redis 253
 redundancy
 clustering for 226
 overview 211
 refactoring abstractions 22
 referential transparency
 benefits of 55–56
 parallel execution 56
 testability 55
 overview 86
 referentially transparent expressions 31, 53
 relational database management system. *See*
 RDBMS
 repeatability 109
 report function 165

repositories
 injecting into service 98–99
 making repository curried argument to service
 methods 100–101
 organizing in modules 98
 reader monad 101–103
 resilience 211
 resilient attribute 32
 resilient models, actors and 210
 responsive to user interaction attribute 32
 REST endpoints 240
 restarting projections 252
 reusability 109
 rich domain models 47–49, 84
 RunnableGraph abstraction 199, 201

S

Scala language 44–71
 ADTs 56–61
 immutability encouraged by 60–61
 pattern matching and 59–60
 structure data in model 58
 sum type and product type 56–57
 making models reactive with 67–71
 effects 67–68
 failures 68–70
 latency 70–71
 modules 62–66
 pure functions for domain behavior 49–56
 benefits of being referentially
 transparent 55–56
 purity of abstractions 53–55
 reasons for using 45–47
 static types and rich domain models 47–49
 scalability 178
 ScalaCheck 55, 268, 274
 scala.concurrent.Future 194–195
 scala.util.Either 159
 scala.util.Try 159
 scalaz.concurrent.Task, using as reactive
 construct 193–195
 scalaz.Free 178
 scalaz.Free.liftF function 174
 scalaz.Validation 159
 sealed trait 86–87
 self-tracing models 40
 Semigroup 137
 separation of concerns 34, 233–237, 284
 CQRS pattern 235–237
 read and write models of persistence 234–235
 sequence combinator 136
 sequence of statements 51
 serializability 163
 service implementations 156

service package 161
 service-level agreement. *See SLA*
 setter 93
 Shapeless 284
 shared mutable state 233
 sharing objects 19
 Show protocol 164–165
 side effects 27–29
 decoupling 28
 in functions 27
 managing 67
 single algebra 82–83
 single trait 62
 Sink, terminating stream to 201–202
 skinny domain objects 84
 SLA (service-level agreement) 35
 Slick 256
 smart constructors idiom 86–88, 104, 119, 158, 271
 Snapshot module 244
 snapshotting API 245
 snapshotting function 252
 solitary look 198
 Source abstraction 199
 Source, setting up 200
 square function 27
 stacking Future and Either 186–187
 state machines 206
 State monad 96, 126, 129–135, 177–178
 statement sequences 51
 stateMonad function 134
 static types 47–49
 step function 247–248
 storage issues 198
 storing changes 237
 stream model 197–205
 back-pressure handling 204–205
 graph as domain pipeline 202–204
 sample use case 198–202
 run computation 202
 setting up Source 200
 split stream into substreams 201
 terminating stream to Sink 201–202
 streams
 splitting into substreams 201
 terminating to Sink 201–202
 structure data in model 58
 Subflow 201
 substitution model 30
 substreams, split streams into 201
 subtyping 60, 124, 129
 Success construct 20
 sum type 56–57, 60, 68, 185
 sumBalances 113
 Summarizer, as PersistentActor 227–228
 supervision, with Akka Streams 225

Supervision.Restart 225
 Supervision.Resume 225
 Supervision.Stop 225

T

Task monad 248
 TaxCalculation 160
 testability 55, 178
 testing domain model 260–278
 algebra of model 267
 designing testable domain models 262–266
 decoupling side effects 263–264
 implementing parametricity and testing 265–266
 providing custom interpreters for domain algebra 264–265
 overview 260–262
 property-based testing 268–277
 data generators 274–277
 modeling properties 268–270
 verifying properties from domain model 270–274
 vs. xUnit-based testing 277
 xUnit-based testing 266–267
 three legs of resiliency 224
 throwing exceptions 20
 timestamped characteristic 239
 traits 12
 trampolined abstraction 195
 transaction netting 217
 TransactionMonoid 201
 transfer function 78–80, 241
 triggered characteristic 239
 Try construct 20
 Try data type 76, 79
 type aliases 154
 Type Class pattern 163–166, 179
 type constructor 115–117, 119–121, 126, 129, 135, 142
 type safety, actors and 207
 type systems 49
 typed actors 123
 types, functional patterns and 143–144

U

ubiquitous language 14–15, 239
 unit method 80
 universally quantified properties 272
 untyped models 215
 update operation 174
 updates 237
 updateState function 244, 250
 util.Try abstraction 22

V

Validation abstraction 123, 136
validation errors 137
validation functions 120, 122, 124, 127, 137–138
validation, fail-fast 138
validCheckingAccountGen 272
value objects 6–9, 11–12, 14, 42
verbosity 99
verifiability 75
verifiable properties 81, 105
verification of correctness 29
verifyId function 262, 264
verifying lens laws 97
versioning 255
Visitor pattern 60
vocabulary 14

W

whileM combinator 133
write model 236

X

xUnit-based testing
general discussion 266–267
overview 55
vs. property-based testing 277

Z

zero operation 110
Zipper 104

MORE TITLES FROM MANNING



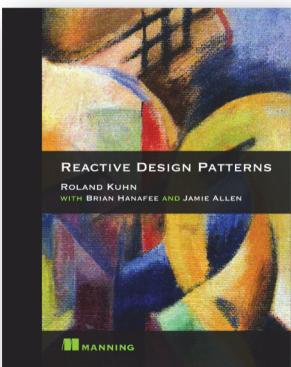
Functional Programming in Scala
by Paul Chiusano and Rúnar Bjarnason

ISBN: 9781617290657

320 pages

\$44.99

September 2014



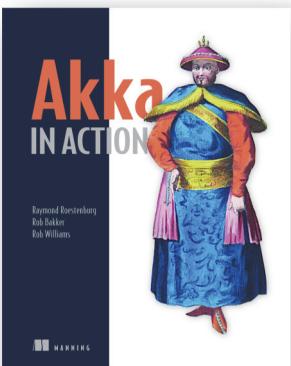
Reactive Design Patterns
by Roland Kuhn
with Brian Hanafee and Jamie Allen

ISBN: 9781617291807

325 pages

\$49.99

December 2016



Akka in Action
by Raymond Roestenburg, Rob Bakker,
and Rob Williams

ISBN: 9781617291012

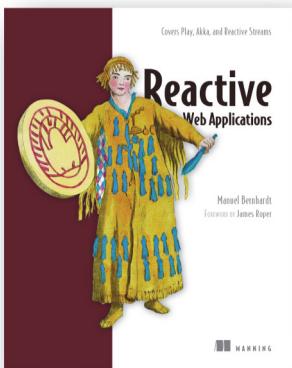
448 pages

\$49.99

September 2016

For ordering information go to www.manning.com

MORE TITLES FROM MANNING



Reactive Web Applications

Covers Play, Akka, and Reactive Streams

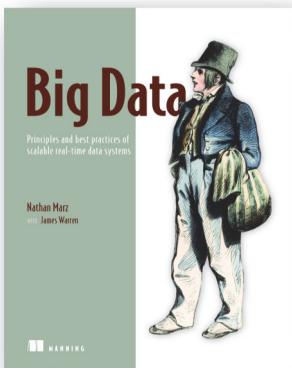
by Manuel Bernhardt

ISBN: 9781633430099

328 pages

\$44.99

June 2016



Big Data

Principles and best practices of scalable realtime data systems

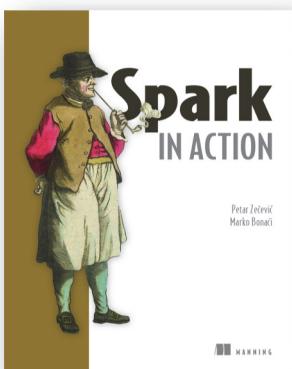
by Nathan Marz and James Warren

ISBN: 9781617290343

328 pages

\$49.99

April 2015



Spark in Action

by Petar Zečević and Marko Bonači

ISBN: 9781617292606

450 pages

\$49.99

October 2016

For ordering information go to www.manning.com

FUNCTIONAL AND REACTIVE DOMAIN MODELING

DEBASISH GHOSH

Traditional distributed applications won't cut it in the reactive world of microservices, fast data, and sensor networks. To capture their dynamic relationships and dependencies, these systems require a different approach to domain modeling. A domain model composed of pure functions is a more natural way of representing a process in a reactive system, and it maps directly onto technologies and patterns like Akka, CQRS, and event sourcing.

Functional and Reactive Domain Modeling teaches you consistent, repeatable techniques for building domain models in reactive systems. This book reviews the relevant concepts of FP and reactive architectures and then methodically introduces this new approach to domain modeling. As you read, you'll learn where and how to apply it, even if your systems aren't purely reactive or functional. An expert blend of theory and practice, this book presents strong examples you'll return to again and again as you apply these principles to your own projects.

WHAT'S INSIDE

- Real-world libraries and frameworks
- Establish meaningful reliability guarantees
- Isolate domain logic from side effects
- Introduction to reactive design patterns

Readers should be comfortable with functional programming and traditional domain modeling. Examples use the Scala language.

Software architect **Debasish Ghosh** was an early adopter of reactive design using Scala and Akka. He's the author of *DSLs in Action*, published by Manning in 2010.

To download their free eBook in PDF, ePUB, and Kindle formats, owners of this book should visit manning.com/books/functional-and-reactive-domain-modeling



“Brings together three different tools—domain-driven design, functional programming, and reactive principles—in a practical way.”

—From the Foreword by Jonas Bonér
Creator of Akka

“A modern approach to implementing domain-driven design.”

—Barry Alexander, Gap

“Debasish confirms his ability to expose DDD practices in a clear way.”

—Cosimo Attanasi, ER Sistemi

“Offers a unique perspective on DDD and FP.”

—Rintcius Blok, Cake Solutions

“A classic in modern software design techniques.”

—William Wheeler
Java/Scala/Akka developer

ISBN-13: 978-1-61729-224-8

ISBN-10: 1-61729-224-9

55999



9 781617 292248



MANNING

US \$ 59.99 | Can \$ 68.99