

grokking
Simplicity

**taming complex software with
functional thinking**

Eric Normand

Foreword by Guy Steele and Jessica Kerr



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

©2021 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
Shelter Island, NY 11964

Development editor: Jenny Stout
Technical development editor: Alain Couniot
Review editor: Ivan Martinović
Production editor: Lori Weidert
Copy editor: Michele Mitchell
Proofreader: Melody Dolab
Technical proofreader: Jean-François Morin
Typesetter: Jennifer Houle
Cover designer: Leslie Haimes

ISBN: 9781617296208

Printed in the United States of America

contents



foreword	xv
preface	xix
acknowledgments	xxi
about this book	xxiii
about the author	xxvii

1 Welcome to Grokking Simplicity	1
What is functional programming?	2
The problems with the definition for practical use	3
The definition of FP confuses managers	4
We treat functional programming as a set of skills and concepts	5
Distinguishing actions, calculations, and data	6
Functional programmers distinguish code that matters when you call it	7
Functional programmers distinguish inert data from code that does work	8
Functional programmers see actions, calculations, and data	9
The three categories of code in FP	10
How does distinguishing actions, calculations, and data help us?	11
Why is this book different from other FP books?	12
What is functional thinking?	13
Ground rules for ideas and skills in this book	14

2	Functional thinking in action	17
.....		
	Welcome to Toni's Pizza	18
	Part 1: Distinguishing actions, calculations, and data	19
	Organizing code by "rate of change"	20
	Part 2: First-class abstractions	21
	Timelines visualize distributed systems	22
	Multiple timelines can execute in different orderings	23
	Hard-won lessons about distributed systems	24
	Cutting the timeline: Making the robots wait for each other	25
	Positive lessons learned about timelines	26
	Part 1: Actions, calculations, and data	29
3	Distinguishing actions, calculations, and data	31
.....		
	Actions, calculations, and data	32
	Actions, calculations, and data apply to any situation	33
	Lessons from our shopping process	36
	Applying functional thinking to new code	39
	Drawing the coupon email process	42
	Implementing the coupon email process	47
	Applying functional thinking to existing code	54
	Actions spread through code	56
	Actions can take many forms	57
4	Extracting calculations from actions	61
.....		
	Welcome to MegaMart.com!	62
	Calculating free shipping	63
	Calculating tax	64
	We need to make it more testable	65
	We need to make it more reusable	66
	Distinguishing actions, calculations, and data	67
	Functions have inputs and outputs	68

Testing and reuse relate to inputs and outputs	69
Extracting a calculation from an action	70
Extracting another calculation from an action	73
Let's see all of our code in one place	85

5 Improving the design of actions **87**

Aligning design with business requirements	88
Aligning the function with business requirements	89
Principle: Minimize implicit inputs and outputs	91
Reducing implicit inputs and outputs	92
Giving the code a once-over	95
Categorizing our calculations	97
Principle: Design is about pulling things apart	98
Improving the design by pulling <code>add_item()</code> apart	99
Extracting a copy-on-write pattern	100
Using <code>add_item()</code>	101
Categorizing our calculations	102
Smaller functions and more calculations	106

6 Staying immutable in a mutable language **109**

Can immutability be applied everywhere?	110
Categorizing operations into reads, writes, or both	111
The three steps of the copy-on-write discipline	112
Converting a write to a read with copy-on-write	113
Complete diff from mutating to copy-on-write	117
These copy-on-write operations are generalizable	118
JavaScript arrays at a glance	119
What to do if an operation is a read and a write	122
Splitting a function that does a read and write	123
Returning two values from one function	124
Reads to immutable data structures are calculations	131
Applications have state that changes over time	132
Immutable data structures are fast enough	133

Copy-on-write operations on objects	134
JavaScript objects at a glance	135
Converting nested writes to reads	140
What gets copied?	141
Visualizing shallow copies and structural sharing	142
7 Staying immutable with untrusted code	147
.....	
Immutability with legacy code	148
Our copy-on-write code has to interact with untrusted code	149
Defensive copying defends the immutable original	150
Implementing defensive copies	151
The rules of defensive copying	152
Wrapping untrusted code	153
Defensive copying you may be familiar with	156
Copy-on-write and defensive copying compared	158
Deep copies are more expensive than shallow copies	159
Implementing deep copy in JavaScript is difficult	160
A dialogue between copy-on-write and defensive copying	162
8 Stratified design: Part 1	167
.....	
What is software design?	168
What is stratified design?	169
Developing our design sense	170
Patterns of stratified design	171
Pattern 1: Straightforward implementations	172
Three different zoom levels	186
Extracting the for loop	189
Pattern 1 Review: Straightforward implementation	198

9 Stratified design: Part 2	201
.....	
Patterns of stratified design	202
Pattern 2: Abstraction barrier	203
Abstraction barriers hide implementations	204
Ignoring details is symmetrical	205
Swapping the shopping cart's data structure	206
Re-implementing the shopping cart as an object	208
The abstraction barrier lets us ignore details	209
When to use (and when <i>not</i> to use!) abstraction barriers	210
Pattern 2 Review: Abstraction barrier	211
Our code is more straightforward	212
Pattern 3: Minimal interface	213
Pattern 3 Review: Minimal interface	219
Pattern 4: Comfortable layers	220
Patterns of stratified design	221
What does the graph show us about our code?	222
Code at the top of the graph is easier to change	223
Testing code at the bottom is more important	225
Code at the bottom is more reusable	228
Summary: What the graph shows us about our code	229
 Part 2: First-class abstractions	 231
 10 First-class functions: Part 1	 233
.....	
Marketing still needs to coordinate with dev	235
Code smell: Implicit argument in function name	236
Refactoring: Express implicit argument	238
Recognize what is and what isn't first-class	240
Will field names as strings lead to more bugs?	241
Will first-class fields make the API hard to change?	242
We will use a lot of objects and arrays	247
First-class functions can replace any syntax	250
For loop example: Eating and cleaning up	253
Refactoring: Replace body with callback	259

What is this syntax?	262
Why are we wrapping the code in a function?	263
11 First-class functions: Part 2	267
.....	
One code smell and two refactorings	268
Refactoring copy-on-write	269
Refactoring copy-on-write for arrays	270
Returning functions from functions	279
12 Functional iteration	289
.....	
One code smell and two refactorings	290
MegaMart is creating a communications team	291
Deriving <code>map()</code> from examples	294
Functional tool: <code>map()</code>	295
Three ways to pass a function	297
Example: Email addresses of all customers	298
Deriving <code>filter()</code> from examples	301
Functional tool: <code>filter()</code>	302
Example: Customers with zero purchases	303
Deriving <code>reduce()</code> from examples	306
Functional tool: <code>reduce()</code>	307
Example: Concatenating strings	308
Things you can do with <code>reduce()</code>	313
Three functional tools compared	315
13 Chaining functional tools	317
.....	
The customer communications team continues	318
Clarifying chains, method 1: Name the steps	324
Clarifying chains, method 2: Naming the callbacks	325
Clarifying chains: Two methods compared	326
Example: Emails of customers who have made one purchase	327
Refactoring existing for loops to functional tools	332

Tip 1: Make data	333
Tip 2: Operate on whole array at once	334
Tip 3: Take many small steps	335
Tip 3: Take many small steps	336
Comparing functional to imperative code	337
Summary of chaining tips	338
Debugging tips for chaining	340
Many other functional tools	341
<code>reduce()</code> for building values	345
Getting creative with data representation	347
Line up those dots	353

14 Functional tools for nested data 355

Higher-order functions for values in objects	356
Making the field name explicit	357
Deriving <code>update()</code>	358
Using <code>update()</code> to modify values	359
Refactoring: replace get, modify, set with <code>update()</code>	361
Functional tool: <code>update()</code>	362
Visualizing values in objects	363
Visualizing nested updates	368
Applying <code>update()</code> to nested data	369
Deriving <code>updateOption()</code>	370
Deriving <code>update2()</code>	371
Visualizing <code>update2()</code> on nested objects	372
Writing <code>incrementSizeByName()</code> four ways	374
Deriving <code>update3()</code>	375
Deriving <code>nestedUpdate()</code>	377
The anatomy of safe recursion	382
Visualizing <code>nestedUpdate()</code>	383
The superpower of recursion	384
Design considerations with deep nesting	386
Abstraction barriers on deeply nested data	387
A summary of our use of higher-order functions	388

15 Isolating timelines 391

There's a bug!	392
Now we can try to click twice fast	393
The timeline diagram shows what happens over time	395
The two fundamentals of timeline diagrams	396
Two tricky details about the order of actions	400
Drawing the add-to-cart timeline: Step 1	401
Asynchronous calls require new timelines	402
Different languages, different threading models	403
Building the timeline step-by-step	404
Drawing the add-to-cart timeline: Step 2	406
Timeline diagrams capture the two kinds of sequential code	407
Timeline diagrams capture the uncertain ordering of parallel code	408
Principles of working with timelines	409
JavaScript's single-thread	410
JavaScript's asynchronous queue	411
AJAX and the event queue	412
A complete asynchronous example	413
Simplifying the timeline	414
Reading our finished timeline	420
Simplifying the add-to-cart timeline diagram: Step 3	422
Review: Drawing the timeline (Steps 1–3)	424
Summary: Drawing timeline diagrams	426
Timeline diagrams side-by-side can reveal problems	427
Two slow clicks get the right result	428
Two fast clicks can get the wrong result	429
Timelines that share resources can cause problems	430
Converting a global variable to a local one	431
Converting a global variable to an argument	432
Making our code more reusable	435
Principle: In an asynchronous context, we use a final callback instead of a return value as our explicit output	436

16 Sharing resources between timelines	441
.....	
Principles of working with timelines	442
The shopping cart still has a bug	443
We need to guarantee the order of the DOM updates	445
Building a queue in JavaScript	447
Principle: Use real-world sharing as inspiration	455
Making the queue reusable	456
Analyzing the timeline	461
Principle: Analyze the timeline diagram to know if there will be problems	464
Making the queue skip	465
17 Coordinating timelines	471
.....	
Principles of working with timelines	472
There's a bug!	473
How the code was changed	475
Identify actions: Step 1	476
Draw each action: Step 2	477
Simplify the diagram: Step 3	481
Possible ordering analysis	483
Why this timeline is faster	484
Waiting for both parallel callbacks	486
A concurrency primitive for cutting timelines	487
Using <code>Cut()</code> in our code	489
Uncertain ordering analysis	491
Parallel execution analysis	492
Multiple-click analysis	493
A primitive to call something just once	500
Implicit versus explicit model of time	503
Summary: Manipulating timelines	507

18 Reactive and onion architectures 509

Two separate architectural patterns	510
Coupling of causes and effects of changes	511
What is reactive architecture?	512
Tradeoffs of the reactive architecture	513
Cells are first-class state	514
We can make <code>ValueCells</code> reactive	515
We can update shipping icons when the cell changes	516
<code>FormulaCells</code> calculate derived values	517
Mutable state in functional programming	518
How reactive architecture reconfigures systems	519
Decouples effects from their causes	520
Decoupling manages a center of cause and effect	521
Treat series of steps as pipelines	522
Flexibility in your timeline	523
Two separate architectural patterns	526
What is the onion architecture?	527
Review: Actions, calculations, and data	528
Review: Stratified design	529
Traditional layered architecture	530
A functional architecture	531
Facilitating change and reuse	532
Examine the terms used to place the rule in a layer	535
Analyze readability and awkwardness	536

19 The functional journey ahead 541

A plan for the chapter	542
We have learned the skills of professionals	543
Big takeaways	544
The ups and downs of skill acquisition	545
Parallel tracks to mastery	546
Sandbox: Start a side project	547
Sandbox: Practice exercises	548
Production: Eliminate a bug today	549

Production: Incrementally improve the design	550
Popular functional languages	551
Functional languages with the most jobs	552
Functional languages by platform	552
Functional languages by learning opportunity	553
Get mathy	554
Further reading	555
 index	 557