

Contents

Preface	vii
1 Introduction	1
1.1 How to use this book.	3
2 The Nature of Complexity	5
2.1 Complexity defined	5
2.2 Symptoms of complexity.	7
2.3 Causes of complexity	9
2.4 Complexity is incremental	11
2.5 Conclusion	11
3 Working Code Isn't Enough	13
3.1 Tactical programming	13
3.2 Strategic programming.	14
3.3 How much to invest?.	15
3.4 Startups and investment	16
3.5 Conclusion	18
4 Modules Should Be Deep	19
4.1 Modular design	19
4.2 What's in an interface?.	20
4.3 Abstractions.	21
4.4 Deep modules	22
4.5 Shallow modules	25
4.6 Classitis.	26
4.7 Examples: Java and Unix I/O.	26

4.8	Conclusion	27
5	Information Hiding (and Leakage)	29
5.1	Information hiding	29
5.2	Information leakage	30
5.3	Temporal decomposition	31
5.4	Example: HTTP server.	32
5.5	Example: too many classes	33
5.6	Example: HTTP parameter handling	34
5.7	Example: defaults in HTTP responses.	36
5.8	Information hiding within a class	37
5.9	Taking it too far	37
5.10	Conclusion	37
6	General-Purpose Modules are Deeper	39
6.1	Make classes somewhat general-purpose	40
6.2	Example: storing text for an editor	40
6.3	A more general-purpose API	41
6.4	Generality leads to better information hiding	43
6.5	Questions to ask yourself.	43
6.6	Conclusion	44
7	Different Layer, Different Abstraction	45
7.1	Pass-through methods	46
7.2	When is interface duplication OK?	47
7.3	Decorators	49
7.4	Interface versus implementation	50
7.5	Pass-through variables	50
7.6	Conclusion	53
8	Pull Complexity Downwards	55
8.1	Example: editor text class	56
8.2	Example: configuration parameters	56
8.3	Taking it too far	57
8.4	Conclusion	58

9	Better Together Or Better Apart?	59
9.1	Bring together if information is shared	60
9.2	Bring together if it will simplify the interface	61
9.3	Bring together to eliminate duplication	61
9.4	Separate general-purpose and special-purpose code	62
9.5	Example: insertion cursor and selection	65
9.6	Example: separate class for logging.	66
9.7	Example: editor undo mechanism.	67
9.8	Splitting and joining methods.	70
9.9	Conclusion	73
10	Define Errors Out Of Existence	75
10.1	Why exceptions add complexity	75
10.2	Too many exceptions.	78
10.3	Define errors out of existence.	79
10.4	Example: file deletion in Windows	79
10.5	Example: Java substring method	80
10.6	Mask exceptions.	81
10.7	Exception aggregation	82
10.8	Just crash?	86
10.9	Design special cases out of existence	87
10.10	Taking it too far	88
10.11	Conclusion	89
11	Design it Twice	91
12	Why Write Comments? The Four Excuses	95
12.1	Good code is self-documenting	96
12.2	I don't have time to write comments	97
12.3	Comments get out of date and become misleading	98
12.4	All the comments I have seen are worthless	98
12.5	Benefits of well-written comments	98
13	Comments Should Describe Things that Aren't Obvious from the Code	101
13.1	Pick conventions.	102
13.2	Don't repeat the code	103

13.3	Lower-level comments add precision	105
13.4	Higher-level comments enhance intuition	107
13.5	Interface documentation	109
13.6	Implementation comments: what and why, not how	116
13.7	Cross-module design decisions	117
13.8	Conclusion	119
13.9	Answers to questions on page 113	120
14	Choosing Names	121
14.1	Example: bad names cause bugs	121
14.2	Create an image	122
14.3	Names should be precise	123
14.4	Use names consistently.	126
14.5	A different opinion: Go style guide	126
14.6	Conclusion	128
15	Write The Comments First	129
15.1	Delayed comments are bad comments.	129
15.2	Write the comments first	130
15.3	Comments are a design tool	131
15.4	Early comments are fun comments	132
15.5	Are early comments expensive?.	132
15.6	Conclusion	133
16	Modifying Existing Code	135
16.1	Stay strategic	135
16.2	Maintaining comments: keep the comments near the code	137
16.3	Comments belong in the code, not the commit log	138
16.4	Maintaining comments: avoid duplication	138
16.5	Maintaining comments: check the diffs	140
16.6	Higher-level comments are easier to maintain	140
17	Consistency	141
17.1	Examples of consistency	141
17.2	Ensuring consistency.	142
17.3	Taking it too far	144

17.4	Conclusion	.144
18	Code Should be Obvious	145
18.1	Things that make code more obvious	.145
18.2	Things that make code less obvious	.148
18.3	Conclusion	.150
19	Software Trends	151
19.1	Object-oriented programming and inheritance	.151
19.2	Agile development.	.153
19.3	Unit tests	.154
19.4	Test-driven development	.155
19.5	Design patterns	.156
19.6	Getters and setters	.156
19.7	Conclusion	.157
20	Designing for Performance	159
20.1	How to think about performance	.159
20.2	Measure before modifying	.161
20.3	Design around the critical path	.162
20.4	An example: RAMCloud Buffers	.163
20.5	Conclusion	.168
21	Conclusion	169
	Index	171
	Summary of Design Principles	175
	Summary of Red Flags	177