



ESSENTIAL SKILLS FOR THE AGILE DEVELOPER

A Guide to Better Programming and Design

ALAN SHALLOWAY
SCOTT BAIN
KEN PUGH
AMIR KOLSKY

*Net*Objectives

Lean-Agile Series

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Praise for *Essential Skills for the Agile Developer*

“I tell teams that the lean and agile practices should be treated like a buffet: Don’t try and take everything, or it will make you ill—try the things that make sense for your project. In this book the authors have succinctly described the ‘why’ and the ‘how’ of some of the most effective practices, enabling all software engineers to write quality code for short iterations in an efficient manner.”

—**Kay Johnson**

Software Development Effectiveness Consultant, IBM

“Successful agile development requires much more than simply mastering a computer language. It requires a deeper understanding of agile development methodologies and best practices. *Essential Skills for the Agile Developer* provides the perfect foundation for not only learning but truly understanding the methods and motivations behind agile development.”

—**R.L. Bogetti**

www.RLBogetti.com,

Lead System Designer, Baxter Healthcare

“*Essential Skills for the Agile Developer* is an excellent resource filled with practical coding examples that demonstrate key agile practices.”

—**Dave Hendricksen**

Software Architect, Thomson Reuters

The Net Objectives Lean-Agile Series

Alan Shalloway, Series Editor



Visit informit.com/netobjectives for a complete list of available publications.

The Net Objectives Lean-Agile Series provides fully integrated Lean-Agile training, consulting, and coaching solutions for businesses, management, teams, and individuals. Series editor Alan Shalloway and the Net Objectives team strongly believe that it is not the software, but rather the value that software contributes—to the business, to the consumer, to the user—that is most important.

The best—and perhaps only—way to achieve effective product development across an organization is a well-thought-out combination of Lean principles to guide the enterprise, agile practices to manage teams, and core technical skills. The goal of **The Net Objectives Lean-Agile Series** is to establish software development as a true profession while helping unite management and individuals in work efforts that “optimize the whole,” including

- The whole organization: Unifying enterprises, teams, and individuals to best work together
- The whole product: Not just its development, but also its maintenance and integration
- The whole of time: Not just now, but in the future—resulting in a sustainable return on investment

The books included in this series are written by expert members of Net Objectives. These books are designed to help practitioners understand and implement the key concepts and principles that drive the development of valuable software.

PEARSON

Essential Skills for the Agile Developer

This page intentionally left blank

Essential Skills for the Agile Developer

A Guide to Better Programming and Design

Alan Shalloway
Scott Bain
Ken Pugh
Amir Kolsky

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

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

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Essential skills for the agile developer : a guide to better programming and design / Alan Shalloway . . . [et al.].

p. cm.

Includes index.

ISBN 978-0-321-54373-8 (pbk. : alk. paper)

1. Agile software development. I. Shalloway, Alan.

QA76.76.D47E74 2011

005.1—dc23

2011023686

Copyright © 2012 Pearson Education, Inc.

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

ISBN-13: 978-0-321-54373-8

ISBN-10: 0-321-54373-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First Printing: August 2011 with corrections made August 2013

To my loving and lifetime partner, Leigh, my muse, who keeps me more humble than I would otherwise be. And while giving me a reason not to be writing books, keeps the pressure up to get the job done.

—Alan Shalloway

*To June Carol Bain. I wish she had lived to see her son become the teacher she always told him he should be.
Hey, mom, you nailed it.*

—Scott Bain

To Ron, Shelly, and Maria: those who matter.

—Amir Kolsky

To my brother Don, who gave me a reason to become an engineer.

—Ken Pugh

This page intentionally left blank

Contents

Series Foreword	xvii
Preface	xxi
Acknowledgments	xxiii
About the Authors	xxv

Part I

The Core Trim Tabs	1
---------------------------	----------

Chapter 1

Programming by Intention	3
Programming by Intention: A Demonstration	3
Advantages	6
Method Cohesion	6
Readability and Expressiveness	7
Debugging	10
Refactoring and Enhancing	11
Unit Testing	13
Easier to Modify/Extend	15
Seeing Patterns in Your Code	16
Movable Methods	17
Summary	18

Chapter 2	
Separate Use from Construction	21
An Important Question to Ask	21
Perspectives	22
Perspective of Creation	23
Perspective of Use	24
What You Hide You Can Change	25
Realistic Approach	27
Other Practical Considerations	30
Timing Your Decisions	30
Overloading and C++	31
Validating This for Yourself	32
Summary	33

Chapter 3	
Define Tests Up Front	35
A Trim Tab: Testing and Testability	35
What Is Testing?	35
Testability and Code Quality	36
Case Study: Testability	37
Setting Ourselves Up for Change	38
Programmer as Frog	39
A Reflection on Up-Front Testing	39
Better Design	42
Improving Clarity of Scope: Avoiding Excess Work	42
Reducing Complexity	42
Other Advantages	43
No Excuses	43
Summary	44

Chapter 4	
Shalloway's Law and Shalloway's Principle	45
Types of Redundancy	46
Copy and Paste	46

Magic Numbers _____	46
Other Types _____	46
Redefining Redundancy _____	46
Other Types of Redundancy _____	47
The Role of Design Patterns in Reducing Redundancy _____	48
Few Developers Spend a Lot of Time Fixing Bugs _____	48
Redundancy and Other Code Qualities _____	50
Summary _____	52

Chapter 5

Encapsulate That! _____ 53

Unencapsulated Code: The Sabotage of the Global Variable _____	53
Encapsulation of Member Identity _____	54
Self-Encapsulating Members _____	56
Preventing Changes _____	58
The Difficulty of Encapsulating Reference Objects _____	59
Breaking Encapsulation with Get() _____	62
Encapsulation of Object Type _____	64
Encapsulation of Design _____	67
Encapsulation on All Levels _____	69
Practical Advice: Encapsulate Your Impediments _____	69
Summary _____	72

Chapter 6

Interface-Oriented Design _____ 75

Design to Interfaces _____	75
Definition of Interface _____	75
Interface Contracts _____	76
Separating Perspectives _____	77
Mock Implementations of Interfaces _____	79
Keep Interfaces Simple _____	79
Avoids Premature Hierarchies _____	80
Interfaces and Abstract Classes _____	81
Dependency Inversion Principle _____	82

Polymorphism in General _____ 83
 Not for Every Class _____ 84
 Summary _____ 84

Chapter 7
Acceptance Test-Driven Development (ATDD) _____ 85

Two Flows for Development _____ 85
 Acceptance Tests _____ 88
 An Example Test _____ 88
 Implementing the Acceptance Tests _____ 90
 User Interface Test Script _____ 90
 User Interface for Testing _____ 91
 XUnit Testing _____ 93
 Acceptance Test Framework _____ 93
 Connection _____ 94
 An Exercise _____ 95
 What to Do If the Customer Won't Tell You _____ 95
 Summary _____ 96

Part II
General Attitudes _____ 97

Chapter 8
Avoid Over- and Under-Design _____ 99

A Mantra for Development _____ 99
 The Pathologies of Code Qualities _____ 100
 Avoid Over- and Under-Design _____ 101
 Minimize Complexity and Rework _____ 102
 Never Make Your Code Worse/Only Degrade Your Code Intentionally __ 102
 Keep Your Code Easy to Change, Robust, and Safe to Change _____ 103
 A Strategy for Writing Modifiable Code in a Non-Object-Oriented or Legacy
 System _____ 103
 Summary _____ 107

Chapter 9	
Continuous Integration _____	109
Branching the Source Code _____	109
Multiple Versions: Specialization Branching _____	110
Working in Isolation: Development Branching _____	112
Problem, Solution, Problem _____	114
The Merge-Back _____	115
Test-Driven Development and Merge Cost _____	117
Continuous Integration _____	119
Continuous Integration Servers _____	121
Summary _____	122

Part III	
Design Issues _____	125

Chapter 10	
Commonality and Variability Analysis _____	127
Using Nouns and Verbs as a Guide: Warning, Danger Ahead! _____	127
What Is the Real Problem? _____	130
What We Need to Know _____	131
Handling Variation _____	132
Commonality and Variability Analysis _____	132
Commonality Analysis _____	132
Variability Analysis _____	133
Object-Oriented Design Captures All Three Perspectives _____	133
A New Paradigm for Finding Objects _____	134
Tips for Finding Your Concepts and Variations with an Example _____	135
The Analysis Matrix: A Case Study _____	136
Selecting the Stories to Analyze _____	141
Summary _____	145

Chapter 11
Refactor to the Open-Closed _____ **147**

The Open-Closed Principle _____ 147

 Open-Closed to Other Things _____ 151

 Open-Closed Is a “Principle” _____ 152

Refactoring _____ 154

 Why Refactor? _____ 155

 Debt versus Investment _____ 155

 Refactoring and Legacy Systems _____ 156

 Refactoring to the Open-Closed _____ 157

 Just-in-Time Design _____ 159

Summary _____ 161

Chapter 12
Needs versus Capabilities Interfaces _____ **163**

The Law of Demeter _____ 163

Coupling, Damned Coupling, and Dependencies _____ 166

 Coupling and Testability _____ 166

 Needs versus Capabilities _____ 167

The Ideal Separation: Needs Interfaces and Capabilities Interfaces _____ 168

Back to the Law of Demeter _____ 169

Summary _____ 171

Chapter 13
When and How to Use Inheritance _____ **173**

The Gang of Four _____ 173

Initial Vectors, Eventual Results _____ 176

Favoring Delegation _____ 178

The Use of Inheritance versus Delegation _____ 180

Uses of Inheritance _____ 181

Scalability _____ 183

Applying the Lessons from the Gang of Four to Agile Development _____ 184

Testing Issues _____ 185

There’s More _____ 187

Part IV Appendixes _____ 189

Appendix A Overview of the Unified Modeling Language (UML) ___ 191

What Is the UML? _____	191
Why Use the UML? _____	192
The Class Diagram _____	192
UML Notation for Access _____	194
Class Diagrams Also Show Relationships _____	194
Showing the “has-a” Relationship _____	195
Composition and Uses _____	195
Composition versus Aggregation _____	196
Notes in the UML _____	196
Indicating the Number of Things Another Object Has _____	197
Dashes Show Dependence _____	198
Sequence Diagram _____	198
Object:Class Notation _____	198
Summary _____	200

Appendix B Code Qualities _____ 201

Christmas-Tree Lights: An Analogy _____	201
Cohesion _____	204
Description _____	204
Principles _____	204
Practices _____	205
Pathologies _____	205
Indications in Testing _____	205
Coupling _____	205
Description _____	205
Principles _____	206
Practices _____	207
Pathologies _____	207
Indications in Testing _____	207

Redundancy _____	207
Description _____	207
Principles _____	208
Practices _____	208
Pathologies _____	208
Indications in Testing _____	208
Encapsulation _____	208
Description _____	208
Principles _____	209
Practices _____	209
Pathologies _____	210
Indications in Testing _____	210

Appendix C	
Encapsulating Primitives _____	211
Encapsulating Primitives in Abstract Data Types _____	211
Principles _____	212
Narrow the Contract _____	213
Expanding Abstract Data Types _____	214
Use Text as External Values _____	215
Enumerations Instead of Magic Values _____	217
Disadvantages _____	218
Summary _____	219
Index _____	221

Series Foreword

The Net Objectives Lean-Agile Series

Alan Shalloway, CEO, Net Objectives

If you are like me, you will just skim this foreword for the series and move on, figuring there is nothing of substance here. You will miss something of value if you do.

I want you to consider with me a tale that most people know but don't often think about. That tale illustrates what is ailing this industry. And it sets the context for why we wrote the Net Objectives Product Development Series and this particular book.

I have been doing software development since 1970. To me, it is just as fresh today as it was four decades ago. It is a never-ending source of fascination to me to contemplate how to do something better, and it is a never-ending source of humility to confront how limited my abilities truly are. I love it.

Throughout my career, I have also been interested in other industries, especially engineering and construction. Now, engineering and construction have suffered some spectacular failures: the Leaning Tower of Pisa, the Tacoma Narrows Bridge, the Hubble telescope. In its infancy, engineers knew little about the forces at work around them. Mostly, engineers tried to improve practices and to learn what they could from failures. It took a long time—centuries—before they acquired a solid understanding about how to do things.

No one would build a bridge today without taking into account long-established bridge-building practices (factoring in stress, compression, and the like), but software developers get away with writing code based on “what they like” every day, with little or no complaint from their peers. And developers are not alone: Managers often require people to work in ways that they know are counterproductive. Why do we work this way?

But this is only part of the story. Ironically, much of the rest is related to why we call this the Net Objectives Product Development Series. The Net Objectives part is pretty obvious. All of the books in this series were written either by Net Objectives staff or by those whose views are consistent with ours. Why product development? Because when building software, it is always important to remember that software development is really product development.

By itself, software has little inherent value. Its value comes when it enables delivery of products and services. Therefore, it is more useful to think of software development as part of product development—the set of activities we use to discover and create products that meet the needs of customers while advancing the strategic goals of the company.

Mary and Tom Poppendieck, in their excellent book *Implementing Lean Software Development: From Concept to Cash* (Addison-Wesley, 2006), note the following:

It is the product, the activity, the process in which software is embedded that is the real product under development. The software development is just a subset of the overall product development process. So in a very real sense, we can call software development a subset of product development. And thus, if we want to understand lean software development, we would do well to discover what constitutes excellent product development.

In other words, software in itself isn't important. It is the value that it contributes—to the business, to the consumer, to the user—that is important. When developing software, we must always remember to look to what value is being added by our work. At some level, we all know this. But so often organizational “silos” work against us, keeping us from working together, from focusing on efforts that create value.

The best—and perhaps only—way to achieve effective product development across an organization is a well-thought-out combination of principles and practices that relate both to our work and to the people doing it. These must address more than the development team, more than management, and even more than the executives driving everything. That is the motivation for the Net Objectives Product Development Series.

Too long, this industry has suffered from a seemingly endless swing of the pendulum from no process to too much process and then back to no process: from heavyweight methods focused on enterprise control to disciplined teams focused on the project at hand. The time has come for management and individuals to work together to maximize

the production of business value across the enterprise. We believe lean principles can guide us in this.

Lean principles tell us to look at the systems in which we work and then relentlessly improve them in order to increase our speed and quality (which will drive down our cost). This requires the following:

- Business to select the areas of software development that will return the greatest value
- Teams to own their systems and continuously improve them
- Management to train and support their teams to do this
- An appreciation for what constitutes quality work

It may seem that we are very far from achieving this in the software-development industry, but the potential is definitely there. Lean principles help with the first three, and understanding technical programming and design has matured far enough to help us with the fourth.

As we improve our existing analysis and coding approaches with the discipline, mind-set, skills, and focus on value that lean, agile, patterns, and Test-Driven Development teach us, we will help elevate software development from being merely a craft into a true profession. We have the knowledge required to do this; what we need is a new attitude.

The Net Objectives Lean-Agile Series aims to develop this attitude. Our goal is to help unite management and individuals in work efforts that “optimize the whole”:

- **The whole organization.** Integrating enterprise, team, and individuals to work best together.
- **The whole product.** Not just its development but also its maintenance and integration.
- **The whole of time.** Not just now but in the future. We want sustainable ROI from our effort.

This Book's Role in the Series

Somewhere along the line, agile methods stopped including technical practices. Fortunately, they are coming back. Scrum has finally acknowledged that technical practices are necessary in order for agility to manifest itself well. Kanban and eXtreme Programming (XP) have

become interesting bedfellows when it was observed that XP had one-piece flow ingrained in its technical practices.

This book was written as a stop-gap measure to assist teams that have just started to do lean, kanban, scrum, or agile. Regardless of the approach, at some point teams are going to have to code differently. This is a natural evolution. For years I have been encouraged that most people who take our training clearly know almost everything they need to know. They just need a few tweaks or a few key insights that will enable them to be more effective in whatever approach they will be using.

Why is this book a “stop-gap measure”? It’s because it is a means to an end. It offers a minimal set of skills that developers need to help them on their way toward becoming adept at incremental development. Once developers master these skills, they can determine what steps they need to take next or what skills they need to acquire next. They are readied for an interesting journey. This book offers the necessary starting point.

The End of an Era, the Beginning of a New Era

I believe the software industry is at a crisis point. The industry is continually expanding and becoming a more important part of our everyday lives. But software development groups are facing dire problems. Decaying code is becoming more problematic. An overloaded workforce seems to have no end in sight. Although agile methods have brought great improvements to many teams, more is needed. By creating a true software profession, combined with the guidance of lean principles and incorporating agile practices, we believe we can help uncover the answers.

Since our first book appeared, I have seen the industry change considerably. The advent of kanban, in particular, has changed the way many teams and organizations do work. I am very encouraged.

I hope you find this book series to be a worthy guide.

—*Alan Shalloway*

CEO, Net Objectives

Achieving enterprise and team agility

Preface

Although this is a technical book, the idea of it sprang from the Net Objectives' agile development courses. As I was teaching teams how to do scrum or lean, students would often ask me, "How are we supposed to be able to build our software in stages?" The answer was readily apparent to me. What they were really asking was, "How can we best learn how to build our software in stages?" I knew of three approaches:

- **Read books.** I am confident that anyone who read and absorbed the books *Design Patterns Explained: A New Perspective on Object-Oriented Design* and *Emergent Design: The Evolutionary Nature of Professional Software Development* would know how to write software in stages.
- **Take courses.** This is a better approach. The combination of Net Objectives courses—Design Patterns and Emergent Design—can't be beat.
- **Learn about trim tabs.** The trim tabs of software development make building software in stages more efficient.

The first one requires a big investment in time. The second one requires a big investment in money. The third one requires less of both. Unfortunately, there is no place where these "trim tabs" are described succinctly.

What are trim tabs? They are structures on airplanes and ships that reduce the amount of energy needed to control the flaps on an airplane or the rudder of a ship. But what I mean comes from something Bucky Fuller once said.

Something hit me very hard once, thinking about what one little man could do.

Think of the Queen Mary—the whole ship goes by and then comes the rudder. And there's a tiny thing at the edge of the rudder called a trim tab.

It's a miniature rudder. Just moving the little trim tab builds a low pressure that pulls the rudder around. Takes almost no effort at all. So I said that the little individual can be a trim tab. Society thinks it's going right by you, that it's left you altogether. But if you're doing dynamic things mentally, the fact is that you can just put your foot out like that and the whole big ship of state is going to go.

So I said, call me Trim Tab.

In other words, these are the actions and insights that give the most understanding with the least investment. In our design patterns courses, we identify three essential trim tabs. Students who do these three things see tremendous improvements in their design and programming abilities. What were these three? Why, they are described in chapters in this book of course:

- Programming by intention
- Separate use from construction
- Consider testability before writing code

These three are very simple to do and take virtually no additional time over not doing them. All three of these are about encapsulation. The first and third encapsulate the implementation of behavior while the second focuses explicitly on encapsulating construction. This is a very important theme because encapsulation of implementation is a kind of abstraction. It reminds us that we are implementing “a way” of doing things—that there may be other ways in the future. I believe forgetting this is the main cause of serious problems in the integration of new code into an existing system.

A fourth trim tab that I recommend is to follow Shalloway's principle. This one takes more time but is always useful.

This book is a compilation of the trim tabs that Net Objectives' instructors and coaches have found to be essential for agile developers to follow to write quality code in an efficient manner. It is intended to be read in virtually any order and in easy time segments. That said, the chapters are sequenced in order to support the flow of ideas.

Acknowledgments

Note from Alan Shalloway

We are indebted to Buckminster Fuller in the writing of this book for many reasons. First, a little bit about Bucky, as he was affectionately known by his friends. I am sorry to say I never met him, but he certainly would have been a dear friend of mine if I had. Bucky was best known for the invention of the geodesic dome and the term “Spaceship Earth.” He also coined the term “synergetics”—the study of systems in transformation—which is essentially what we do at Net Objectives. Of course, most relevant is that his use of the term “trim tab” (discussed in the preface) was the actual inspiration for this book.

He was also an inspiration for me to always look for better ideas. This quote is my all-time favorite Buckyism:

I am enthusiastic over humanity's extraordinary and sometimes very timely ingenuity. If you are in a shipwreck and all the boats are gone, a piano top buoyant enough to keep you afloat that comes along makes a fortuitous life preserver. But this is not to say that the best way to design a life preserver is in the form of a piano top. I think that we are clinging to a great many piano tops in accepting yesterday's fortuitous contrivings as constituting the only means for solving a given problem.

All these are good reasons, of course. But in truth, I realized I wanted to make a special acknowledgment for Bucky because he has been an inspiration in my life from, ironically, mostly the moment he passed away in 1983. He was not just one of these vastly intelligent men or one of these great humane folks. He was a rare, unique combination of both. If you are not familiar with this great man, or even if you are, I suggest you check out the Buckminster Fuller Institute (<http://www.bfi.org>).

We Also Want to Acknowledge

This book represents our view of those skills that we believe every agile software developer should possess. However, we did not come up with this guidance on our own, and we owe a debt of sincere gratitude to the following individuals.

Christopher Alexander, master architect and author of *The Timeless Way of Building*. Although he is not a technical expert, Alexander's powerful ideas permeate nearly all aspects of our work, most especially the concept "design by context."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software*. Although we hope to have significantly advanced the subject of their work, it was the genesis of much of the wisdom that guides us today.

James Coplien wrote the thesis "Multi-Paradigm Design" that became the book that taught us about Commonality-Variability Analysis. This in turn helped us understand how to use patterns and objects in a way that fits the problem domain before us. Jim's work is a powerful enabler of many of the skills we teach in this book.

Martin Fowler, author of *Refactoring* and *UML Distilled*, as well as many other thoughtful and incredibly useful books. Martin is definitely the developer's friend.

Ward Cunningham, one of the author/inventors of eXtreme Programming and the progenitor of the role of testing in the daily life of the software developer. Countless good things have come from that central idea. Also, Ward, thanks so much for inventing wikis.

Robert C. Martin, author of *Agile Software Development* and many other books and articles. "Uncle Bob" teaches how various critical coding skills work together to make software that is readable, scalable, maintainable, and elegant.

In addition to these individual authors and thought leaders, we also want to acknowledge the thousands of students and consulting clients who have contributed endlessly to our understanding of what good software is and how to make it. It has been said that the good teacher always learns from the student, and we have found this to be true to an even greater degree than we expected when Net Objectives was founded more than 10 years ago. Our clients have given us countless opportunities to expand our thinking, test our ideas, and gain critical feedback on their real-world application.

There would be no Net Objectives without our customers. We love our customers.

About the Authors



Alan Shalloway is the founder and CEO of Net Objectives. With more than 40 years of experience, Alan is an industry thought leader in lean, kanban, product portfolio management, scrum, and agile design. He helps companies transition to lean and agile methods enterprisewide as well teaches courses in these areas. Alan has developed training and coaching methods for lean-agile that have helped Net Objectives' clients achieve long-term, sustain-

able productivity gains. He is a popular speaker at prestigious conferences worldwide. He is the primary author of *Design Patterns Explained: A New Perspective on Object-Oriented Design* and *Lean-Agile Pocket Guide for Scrum Teams*. Alan has worked in dozens of industries over his career. He is a cofounder and board member for the Lean Software and Systems Consortium. He has a master's degree in computer science from M.I.T. as well as a master's degree in mathematics from Emory University. You can follow Alan on Twitter @alshalloway.



Scott Bain is a 35+-year veteran in computer technology, with a background in development, engineering, and design. He has also designed, delivered, and managed training programs for certification and end-user skills, both in traditional classrooms and via distance learning. Scott teaches courses and consults on agile analysis and design patterns, advanced software

design, and sustainable Test-Driven Development. Scott is a frequent speaker at developer conferences such as JavaOne and SDWest. He is the author of *Emergent Design: The Evolutionary Nature of Professional Software*

Development, which won a Jolt Productivity Award and is now available from Addison-Wesley.



Ken Pugh is a fellow consultant with Net Objectives. He helps companies transform into lean-agility through training and coaching. His particular interests are in communication (particularly effectively communicating requirements), delivering business value, and using lean principles to deliver high quality quickly. He also trains, mentors, and testifies on technology topics ranging from object-oriented design to Linux/Unix. He has written several programming books, including the 2006 Jolt Award winner, *Prefactoring: Extreme Abstraction, Extreme Separation, Extreme Readability*. His latest book is *Lean-Agile Acceptance Test Driven Development: Better Software Through Collaboration*. He has helped clients from London to Boston to Sydney to Beijing to Hyderabad. When not computing, he enjoys snowboarding, windsurfing, biking, and hiking the Appalachian Trail.



Amir Kolsky is a senior consultant, coach, and trainer for Net Objectives. Amir has been in the computer sciences field for more than 25 years. He worked for 10 years in IBM Research and spent 9 more years doing chief architect and CTO work in assorted companies big and small. He has been involved with agile since 2000. He founded MobileSpear and subsequently XPand Software, which does agile coaching, software education, and agile projects in Israel and Europe. Amir brings his expertise to Net Objectives as a coach and trainer in lean and agile software processes, tools, and practices, Scrum, XP, design patterns, and TDD.

This page intentionally left blank

CHAPTER 8

Avoid Over- and Under-Design

Developers tend to take one of two approaches to programming. Many think they need to plan ahead to ensure that their system can handle new requirements that come their way. Unfortunately, this planning ahead often involves adding code to handle situations that never come up. The end result is code that is more complex than it needs to be and therefore harder to change—the exact situation they were trying to avoid. The alternative, of course, seems equally bad. That is, they just jump in, code with no forethought, and hope for the best. But this hacking also typically results in code that is hard to modify. What are we supposed to do that doesn't cause extra complexity but leaves our code easy to change? The middle ground can be summed up by something Ward Cunningham said at a user group: "Take as much time as you need to make your code quality as high as it can be, but don't spend a second adding functionality that you don't need now!" In other words, write high-quality code, but don't write extra code.

*This chapter is admittedly more of a new mantra than it is a detailed description of a technique to implement. This chapter takes advantage of what we learned in Chapter 5, *Encapsulate That!*, and sets the groundwork for Chapter 11, *Refactor to the Open-Closed*.*

A Mantra for Development

We believe developers should have a particular attitude when writing code. There are actually several we've come up with over time—all being somewhat consistent with each other but saying things a different way. The following are the ones we've held to date:

- Avoid over- and under-design.
- Minimize complexity and rework.

- Never make your code worse (the Hippocratic Oath of coding).
- Only degrade your code intentionally.
- Keep your code easy to change, robust, and safe to change.

Before we can discuss these mantras, we need to be clear what we mean by quality code. Appendix B, Code Qualities, provides a thorough explanation of the specific qualities referred to in this chapter. We'll give a brief summary of code quality here, but interested readers may want to read the more extensive narrative in the appendix.

The Pathologies of Code Qualities

It's often easier to see code qualities by discussing examples of when the qualities aren't present. Let's look at five common code qualities: cohesion, coupling, redundancy, readability, and encapsulation.

- **Cohesion.** Strongly cohesive classes are classes whose functions are all related to each other. Strongly cohesive methods are methods that do only one thing. The pathology of weak cohesion is classes or methods that do unrelated things. We've heard very weakly cohesive classes called "god objects" presumably because they are somewhat omniscient in that everything takes place in them.¹
- **Proper coupling.** Having well-defined relationships between objects makes them easier to understand and likely to inadvertently cause problems when changing code. The pathology of improper coupling is the occurrence of side effects—that is, unexpected errors due to making changes elsewhere.
- **No redundancy.** No redundancy is difficult to achieve. The more redundancy you have, the more time it will take to make changes. As we discussed in Chapter 4, Shalloway's Law and Shalloway's Principle, no redundancy is virtually impossible to achieve—but at least you want to make it so you don't have to find the duplication. Essentially, the pathology of redundancy is that when you make a change in one place, you have to make a change in another place.

1. We've also thought they may be called this because when you first look at them you mutter to yourself "Oh my god!" and the fact that it looks like only god could figure them out.

- **Readability.** Readable code means you can understand what has been written. It requires intention-revealing names and is best achieved by using Programming by Intention (see Chapter 1, Programming by Intention). Unreadable code, of course, is code you can't understand when you read it. Poor names, tight coupling, and big methods/classes contribute greatly to the unreadability of code.
- **Encapsulation.** Encapsulation is more than mere data hiding. The type of an object is one of the most important things to hide. Design patterns are really about hiding: object type, cardinality, which function is being used, order, optional behavior, construction, and more. The pathology of encapsulation is when you must know how the code you are using is implanted in order to use it properly. This often means you know the implementation type of the object being used or know something about cardinality, order, and so on.

Avoid Over- and Under-Design

This essentially means you should put in the correct amount of design. Overdesign is putting in things that add complexity to the code that may or may not be needed. Note that the key word here is “complexity.” We're not as worried about the time you take as much as we are about how you leave the state of the code. If the work you've done does not raise the complexity of the code you have, then no worries. In other words, putting in an interface where one may or may not be needed is not necessarily a bad thing if everyone understands interfaces. Interfaces aren't really complexity-adders in our mind. They are a holder for an idea. However, putting in a complex parameter list (or using a value object to hold a parameter, say, when one isn't needed) would be raising complexity.

Under-design is actually a euphemism for “poor code quality.” We view under-design as having taken place when high coupling or weak cohesion is present. Typically, proper encapsulation is also not present. So, avoiding overdesign means make your code changeable, but don't add things you don't need now. If you need them later, the changeability of the code will enable you to do that with less, if any, extra cost. Avoiding under-design mostly means making sure your code is changeable.

Minimize Complexity and Rework

Many people only partly understand the true nature of refactoring. Martin Fowler, in his excellent *Refactoring: Improving the Design of Existing Code*,² describes refactoring in the following way.

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.

In the book, Fowler talks about refactoring as a method of cleaning up messy/poor code. However, there is another side to refactoring that Fowler doesn't talk about. This is refactoring code that is of high quality, when it comes to the code qualities we've been talking about, but that no longer has sufficient design because of new requirements. In other words, the book talks about how to clean up poorly written code (a good thing to know) but mostly ignores how to refactor good code that now must be changed to accommodate new requirements.³

We strongly suggest that refactoring good code when new requirements come so that the code is better able to accommodate the changes is a way to minimize complexity because you are deferring adding complexity until it is needed, but your code quality is high so there is no rework. We would contend that delaying extensions to code is not rework but a kind of just-in-time design. We'll talk explicitly about how to do this in Chapter 11, Refactor to the Open-Closed.

Never Make Your Code Worse/Only Degrade Your Code Intentionally

Existing code degrades one bit at a time (no pun intended). We suggest that team members do their best to not take shortcuts that makes their code worse. Sometimes this is difficult, however. It may be that legacy code makes it very difficult to add functionality properly without harming your code. To be realistic, we restate "Never make your code worse"

2. Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 1999.

3. Alan Shalloway had a private conversation with Martin about this once. After suggesting that the refactoring concepts Martin presented would work equally well for both types of code, Martin responded by agreeing and saying, "My book was long enough as it was!"

to “Only degrade your code intentionally.” Although this may sound funny, the alternative would be to make your code worse *unintentionally*.

One way to only degrade your code intentionally is to ensure you consider alternatives. One way to do this is to make a teamwide agreement that if developers can’t figure out how to make a change without degrading code, they will tell another team member of the change they are thinking of making before they make the change. Note that we are not requiring getting permission or even getting a better result. We’re just suggesting you tell someone. This forces you to at least reflect a little. Our experience has shown us that a person will stop just short of a good solution because he or she is willing to do the first thing that comes to mind. Our approach forces people to think about things a bit more (sometimes a lot more because they don’t want to admit to coworkers that they don’t have good solutions).

Keep Your Code Easy to Change, Robust, and Safe to Change

Code should not be viscous. That means the effort to make changes should not be excessive. Viscosity can be avoided by having easy-to-understand, nonredundant code. Code should also not be brittle. That is, changes in one place should not break code in other places. This requires loosely coupled code, following Shalloway’s principle (see Chapter 4, Shalloway’s Law and Shalloway’s Principle) and proper encapsulation. It is not sufficient to follow these two mantras alone, however. Although doing so may make it easy to change your code with less likelihood of breaking it, there are no guarantees. The only way to be assured that you can safely change your code is to have a full set of automated acceptance tests available.

A Strategy for Writing Modifiable Code in a Non-Object-Oriented or Legacy System

Many of the approaches we’ve discussed here are often met with this attitude: “That’s a great idea, but I can’t do it where I work because I’m using C.” A variant of this is “That’s a great idea, but I can’t do it where I work because there is so much monolithic legacy code that I can’t take advantage of object-oriented methods.” There are other variants as well, but you get the idea. Although it is true that your existing software and

the languages you are using provide certain constraints on what you can do, there are certain approaches you can *always* take. One of these is to consider the separation of concerns in a different way.

The idea is to separate the code that is particular to the application from the code that defines the application's architecture (or even system architecture).

One can think of a program as essentially an overall flow detailing the steps to be undertaken. For example, a sales-order system can have a variety of actions needed to work:

- Select customer.
- Get customer information.
- Select products to be sold.
- Get prices.
- Apply appropriate discounts.
- Total cost of sales order.
- Specify shipping.

Object orientation attempts to simplify this by creating objects that group responsibilities for the different implementing steps. These objects collaborate with each other and avoid coupling by having well-defined interfaces that hide their implementations. Unfortunately, if you can't (properly) use an object-oriented language, how can you get at least some of the value that comes from separating concerns? One way is to have each method in your code deal with only one of the following:

- The system architecture
- The application architecture
- The implementation of a step

For example, let's say you are writing embedded software that takes its input from a special bus in the form of string from which it extracts required parameters via a specialized method. Applications like this often take the following approach:

```
public function someAction () {
    string inputString;
```

```

inputString= getInputFromBus();
if (getParameter(inputString, PARAM1)> SOMEVALUE) {
    // bunches of code
} else {
    if (getParameter(inputString, PARAM2)< SOMEOTHERVALUE) {
        // more bunches of code
        // ...
    } else {
        // even more bunches of code
        // ...
    }
}
}
}

```

The problem with this is lack of cohesion. As you try to figure out what the code does, you are also confronted with detailed specifics about how the information is obtained. Although this might be clear to the person who first wrote this, this will be difficult to change in the future (not counting the confusion that happens now). This gets much worse if one never makes the distinction between the system one is embedded in (which is determining the input method) and the logic inside the routine. For example, consider what happens when a different method of getting the string is used as well as a different method of extracting the information. In this case, the parameters are returned in an array:

```

public function someAction () {
    string inputString;
    int values[MAX_VALUES];

    if (COMMUNICATION_TYPE== TYPE1) {
        inputString= getInputFromBus();
    } else {
        values= getValues();
    }

    if ( (COMMUNICATION_TYPE== TYPE1 ?
        getParameter( inputString, PARAM1) :
        values[PARAMETER1]) > SOMEVALUE) {
        // bunches of code
    } else {
        if ( COMMUNICATIONS_TYPE== TYPE1 ?
            getParameter( inputString, PARAM2) :
            values(PARAMETER2])
            < SOMEOTHERVALUE) {
            // more bunches of code
            // ...
        }
    }
}

```

```

    } else {
        // even more bunches of code
        // ...
    }
}
}
}

```

Pretty confusing? Well, have no fears, it'll only get worse. If, instead, we separated the “getting of the values” from the “using of the values,” things would be much clearer.

```

public function someAction () {
    string inputString;
    int values[MAX_VALUES];

    int value1;
    int value2;

    if (COMMUNICATION_TYPE== TYPE1) {
        inputString= getInputFromBus();
    } else {
        values= getValues();
    }

    value1= (COMMUNICATION_TYPE== TYPE1 ?
        getParameter( inputString, PARAM1) :
        values[PARAMETER1]);
    value2= ( COMMUNICATIONS_TYPE== TYPE1 ?
        getParameter( inputString, PARAM2) :
        values(PARAMETER2]);

    someAction2( value1, value2);
}

public function someAction2 (int value1, int value2) {

    if ( value1 > SOMEVALUE) {
        // bunches of code
    } else {
        if ( value2 < SOMEOTHERVALUE) {
            // more bunches of code
            // ...
        } else {
            // even more bunches of code
            // ...
        }
    }
}
}
}

```

You must remember that complexity is usually the result of an increase in the communication between the concepts involved, not the concepts themselves. Therefore, complexity can be lowered by separating different aspects of the code. This does not require object orientation. It simply requires putting things in different methods.

Summary

Developers must always be aware of doing too much or too little. When you anticipate what is needed and put in functionality to handle it, you are very likely to be adding complexity that may not be needed. If you don't pay attention to your code quality, however, you are setting yourself up for rework and problems later. Code quality is a guide. Design patterns can help you maintain it because they give you examples of how others have solved the problem in the past in similar situations.

This page intentionally left blank

Index

A

Abstract classes

- creating one-to-one relationship, 52
- definition of, 194–195
- in encapsulation of object type, 65
- interfaces as, 81–82, 195
- reducing redundancy using, 48
- specification giving better understanding of, 134
- in Variability Analysis, 133–134

Abstract data types (ADTs)

- cost of not using, 219
- disadvantages of, 219
- encapsulating primitives in, 211–212
- enumerations instead of magic values, 218–219
- expanding, 214
- narrowing contract for concept, 213
- underlying principles of, 212
- using text as external values, 215–217

Abstract inheritance, 175–176, 179

Acceptance Test-Driven Development (ATDD)

- acceptance test framework, 93–94
- acceptance tests, 88
- benefits of, 40
- combining unit tests for automated, 39
- connection, 94–95
- creating unit tests from, 40, 44
- defined, 41, 85
- example test, 88–89
- exercise, 95
- flows for development, 85–87
- implementing with FIT, 40
- improving clarity of scope, 42
- no excuses for avoiding, 43
- other advantages, 43
- reducing complexity, 42
- role in continuous integration, 119–120
- summary review, 96
- testing interface to clarify contract, 76–77
- user interface for testing, 91–92
- user interface test script, 90–91
- UTDD vs., 42
- what to do if customer won't tell you, 95–96
- XUnit testing, 93

Accessibility, UML notation for, 194

Accessor methods

- encapsulating data members behind set of, 56–57
- preventing changes, 58–59

Accidental coupling, 206

Adapter Pattern, 168, 171

ADTs. *See* Abstract data types (ADTs)

Aggregation, UML

- composition vs., 196
- example of, 193
- showing "has-a" relationship, 195

Agile Software Development (Martin), xxiv

Alexander, Christopher, 40, 42

Aliasing, 60

Analysis Matrix

- adding another case, 139–140
- adding new concepts as rows in table, 139–141
- adding steps, 137–139
- building, 136–137
- defined, 136
- selecting stories to analyze, 141–144
- uses of, 144–145

"Anticipated vector of change," 57

ATDD. *See* Acceptance Test-Driven Development (ATDD)

Attributes, abstract data type, 214

Automated tests, in continuous integration, 119–120

B

Bain, Scott, xxv–xxvi

Bain's corollary, 51

Behavior

- delegating to helper class, 80–81
- designing from context of, 40–42
- determining functionality using tests, 36
- enhancing by adding or changing, 11
- preserving in refactoring, 11

redundant, 46

timing decisions and, 30–31

unit testing of, 13–15

Branching source code

- development branching, 112–114
- merge-back and, 115–117
- overview of, 109–110
- problem, solution, problem and, 114
- specialization branching, 110–111
- version control and, 109

C

C++

- abstract data types in, 218
- Commonality-Variability Analysis and, 135
- composition vs. aggregation in, 196
- overloading and, 31–32
- separating implementation from interface, 84
- testing private methods, 14

C#

- abstract data types in, 218
- breaking encapsulation with `get()`, 63–64
- Commonality-Variability Analysis and, 134–135
- composition vs. aggregation in, 196
- declaring interface in, 76
- encapsulation of object type in, 66
- global variables in, 53–54
- interfaces and abstract classes in, 82
- mixing implementation with interface in, 84
- new keyword in, 23
- self-encapsulating data members in, 59

Cable, continuous integration server, 121

Capabilities interface

- defined, 168
- following Law of Demeter, 169–171

- needs vs. capabilities and, 167–168
 - separating from needs interface, 168–169
- Cardinality of relationship, UML, 197
- Case studies
 - Analysis Matrix. *See* Analysis Matrix
 - testability, 37–38
- Christmas-tree light analogy, of code quality, 201–204
- CI. *See* Continuous integration (CI)
- Clarity, using UML for, 192
- Class diagrams, UML
 - indicating number of things another object has, 197
 - overview of, 192–193
 - showing composition and uses relationship, 195–196
 - showing "has-a" relationship, 195–196
 - showing relationships, 194–195
 - using notes, 196
- Classes
 - abstract data types increasing number of, 219
 - changing from concrete to abstract, 25–26
 - cohesion of, 6–7
 - designing in object-oriented analysis, 127
 - improving cohesion with movable methods, 17–18
 - interfaces and abstract, 81–82
 - keeping interfaces simple, 79–80
 - pathologies of cohesion and, 205
 - separate interface declarations and, 84
 - showing relationships, in UML diagrams, 194–195
 - testability case study, 37–38
 - warning about nouns and, 129–130
- Classes, cohesive, 204–205
- Clients
 - breaking encapsulation with `get()`, 62–64
 - coupling and testability, 166–167
 - encapsulating reference objects, 59–61
 - Law of Demeter and, 169–171
 - separate needs/capabilities interfaces, 167–169
- Code qualities
 - avoiding slow degradation of, 39
 - Christmas-tree light analogy, 201–204
 - cohesion, 100, 204–205
 - coupling, 100, 205–207
 - encapsulation, 101, 208–210
 - overview of, 201
 - readability, 101
 - redundancy, 50–52, 100, 207–208
 - testability, 36–37
- Code smells
 - comments as, 8–9
 - identifying need for refactoring, 155
- Cohesion
 - Christmas-tree light analogy, 203
 - as code quality, 100, 204–205
 - creating with abstract data types, 212
 - under-design creating weak, 101
 - encapsulation and, 209
 - method, 6–8
 - movable methods and, 17–18
 - nouns-and-verbs approach causing weak, 130–131
 - testability related to, 36–38
 - writing modifiable code for legacy systems and, 105
- Comments
 - as code smell, 8–10
 - Programming by Intention vs. using, 7–8
- `commit()` method, 13–14, 16–17
- Commonality Analysis, 132–133

- Commonality-Variability Analysis
 - case study: The Analysis Matrix, 136–141
 - of cohesion, 205
 - Commonality Analysis, 132–133
 - finding objects, 134–136
 - handling variation, 132
 - nouns and verbs and, 127–130
 - overview of, 127
 - perspectives, 133–134
 - the real problem, 130–131
 - selecting stories to analyze, 141–145
 - summary review, 145
 - Variability Analysis, 133
 - what we need to know, 131–132
- Communication, UML for, 192
- Complexity
 - lowering by separating aspects of code, 107
 - of merge process in development branching, 112–113
 - minimizing rework and, 99, 102
 - overdesign creating, 101
 - reducing with ATDD, 42–43
 - refactoring and enhancing to add, 11–12
- Composition, UML
 - example of, 193
 - showing "has-a" relationship, 195
 - showing uses relationship, 195–196
 - and uses, 195–196
 - vs. aggregation, 196
- Construction, separating use from
 - approach to, 27–30
 - coupling and, 25–27
 - debugging and, 48–50
 - perspective of, 23–24
 - question to ask, 21–22
 - realistic approach to, 27–30
 - timing decisions, 30–31
 - validating concept, 32–33
- Constructors
 - C++ and overloaded, 31–32
 - difficulty of encapsulating reference objects, 59–61
 - private, 30
- Context object, encapsulation of design, 68–69
- Continuous integration (CI)
 - branching source code, 109–110
 - development branching, 112–114
 - merge-back, 115–117
 - "nightly build," 115
 - overview of, 109
 - problem, solution, problem, 114–115
 - servers, 121–122
 - specialization branching, 110–111
 - summary review, 122–123
 - text-driven development and merge cost, 117–119
 - understanding, 119–120
- Contracts, interface, 76–77
- Copy and paste, as redundancy, 46
- Cost of merge
 - in development branching, 112–114
 - merge-backs decreasing, 116–117
 - "nightly builds" increasing, 115
 - problem, solution, problem and, 114
 - test-driven development and, 117–119
- Coupling
 - Christmas-tree light analogy and, 203
 - as code quality, 100, 205–207
 - under-design creating high, 101
 - eliminating to implementations, 166–168
 - encapsulation and, 54–56, 209
 - global variables creating tight, 54
 - identity, 54, 206
 - inheritance, 206
 - misuse of inheritance and, 176–178
 - nouns-and-verbs approach causing tight, 130–131
 - by perspective, 23–26
 - redundancy related to, 47, 50–52

- representational, 206
- specialization branching and, 110–111
- subclass, 206
- testability and, 36–38, 166–167

CruiseControl, continuous integration server, 121

D

Dashes, showing dependence in UML, 198

Data hiding. *See* Encapsulation

Debt vs. investment, refactoring, 156

Debugging

- finding bugs, 48–50
- Programming by Intention and, 10
- refactoring vs., 154

Degrade code intentionally, 100, 102–103

Delegation

- avoiding premature hierarchies, 80–81
- favoring, 178–179
- Gang of Four and, 173–176
- inheritance vs., 180–181

Dependence, using dashes in UML, 198

Dependencies

- controlling with strong interfaces, 163
- coupling and, 205–207
- Law of Demeter and, 163–165

Dependency injection, 28

Dependency Inversion principle, 42, 82

Deserialization, 28

Design

- of code for testability, 37
- encapsulation of, 67–69
- lessons from Gang of Four, 184–185
- starting with big picture of wanted behavior, 40
- testability case study, 38
- up-front testing as up-front, 39–40
- using tests to accomplish, 36

Design, avoid over- and under-code qualities and, 100–101

- degrading code intentionally, 102–103
- development mantras, 99–100
- keeping code robust and easy/safe to change, 102–103
- minimizing complexity and rework, 102
- modifying code for non-object-oriented or legacy system, 103–107
- overview of, 101

Design patterns

- concept of, 42
- encapsulating using, 70–71
- reducing redundancy with, 48
- seeing in code, 16–17

Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Erich, et al.), 48, 173, xxiv

Design Patterns Explained: A New Perspective on Object-Oriented Design (Shalloway and Trott), 48, 67–68, 100, xxv

Design Patterns Repository, 71

Development branching, 110, 112–114

Diagrams, UML

- class, overview of, 192–193
- class, showing relationships, 194–195
- defined, 191
- sequence, 198

Direct inheritance, 174–176

E

Emergent Design: The Evolutionary Nature of Professional Software Development (Bain), xxv–xxvi

Encapsulation

- on all levels, 69
- breaking, with `get()`, 62–64

Encapsulation (*continued*)

- as code quality, 101, 208–210
 - coupling from perspective of, 25–27
 - of design, 67–69
 - in development branching, 112
 - importance of, 22
 - of member identity, 54–56
 - of object type, 64–66
 - overview of, 53
 - in practice, 69–72
 - preventing changes, 58–59
 - of reference objects, 59–61
 - removing need for specialization
 - branching, 111
 - for robust and easy/safe to change code, 103
 - self-encapsulating members, 56–57
 - summary review, 72–73
 - testability related to proper, 36–37
 - unencapsulated code, 53–54
- Encapsulation of primitives
- in abstract data types, 211–212
 - disadvantages of, 219
 - enumerations instead of magic values, 218–219
 - expanding abstract data types, 214
 - narrowing contract for concept, 213
 - overview of, 211
 - principles, 212
 - using text as external values, 215–217
- Enhancing systems
- with Programming by Intention, 12
 - refactoring vs., 154
- Enumerations, encapsulating magic values with, 218–219
- Expressiveness, in Programming by Intention, 7–10
- Extensions, code
- delaying, 102
 - Programming by Intention and, 15–16
- External values, using text as, 215–217
- Extract Method, of refactoring, 11

F

- Façade Pattern, 79, 171
- Factory
- adding to build instance, 150–151
 - encapsulation of design, 67–69
 - practical considerations, 30
 - separating use from construction, 26–28
- Failure, Christmas-tree light analogy, 201–204
- Failure report, interfaces, 77
- Feature envy, refactoring, 155
- FinalBuilder, continuous integration server, 121
- FIT for Developing Software* (Mugridge), 40
- FIT (Framework for Integrated Testing), 40
- Framework for Integrated Testing (FIT), 40
- Frogs, programmers as, 39
- Fuller, Buckminster, xxi–xxiii
- Functional steps, of programming languages, 4–5
- Functions
- designing in object-oriented analysis, 127
 - Law of Demeter for, 164–165
 - limited in ADTs, 213
 - warning about verbs, 129–130

G

- Gang of Four
- applying lessons to agile development, 184–185
 - big design up-front, 72
 - favoring delegation over inheritance, 80, 178–179
 - favoring design to interface, 66, 75
 - on inheritance, 173–176
 - on using inheritance to hide variation, 181–182

`getInstance()` method, 28–32
`get()` method
 breaking encapsulation with, 62–64
 difficulty of encapsulating reference objects, 59–61
 encapsulating data members behind, 56–57
 preventing changes, 58–59
`getX()` method
 difficulty of encapsulating reference objects, 59–61
 encapsulation of member identity, 56
 preventing changes, 58
 self-encapsulating data members, 57
 Global variables, unencapsulated code and, 53–54

H

"Has-a" relationship, UML class diagram, 192, 195–196
 Helper methods
 debugging by examining, 10
 defined, 5
 development of, 147
 enhancing system by changing, 12
 Open-Closed principle and, 153
 seeing patterns in code, 16–17
 unit testing behavior of, 13–14
 Hierarchies
 avoiding premature, 80–81
 causes of tall class, 127–128
 creating good, decoupled, 128
 inheritance, 177

I

Identity coupling, 54, 206
 Identity, encapsulation of data member, 54–56, 64–66
`if` statement, 127, 129–130
 "Itly" tests, for interfaces, 77

Impediments, encapsulating your, 69–71
 Implementation perspective
 abstract classes and, 82
 in interface-oriented design, 75
 separating specification perspective from, 77–79, 84
 Implementations
 eliminating coupling to, 168
 encapsulating in abstract data types. *See* Encapsulation of primitives
 redundant, 46, 47
 removing redundant, 48
Implementing Lean Software Development: From Concept to Cash (Poppendieck), xviii
 Inheritance
 delegation vs., 80–81, 178–181
 Gang of Four and, 173–176, 184–185
 improper use of, 130–131, 173
 initial vectors and eventual results, 176–178
 open-closed through direct, 148–149
 scalability and, 183–184
 specialization branching similar to, 110
 testing issues, 185–186
 uses of, 181–183
 when and how to use, 173
 Inheritance coupling, 206
 Initial vectors, eventual results and, 176–178
 Instances. *See* Construction, separating use from; Use, separating construction from
 Intention-revealing names
 and cohesion, 205
 defined, 8
 for interfaces, 77
 for readability, 101
 Intentional coupling, 206
 Interaction diagrams, UML, 198–199

Interface contracts, 76–77
 interface keyword, 76
 Interface-oriented design (IOD)
 abstract classes and, 81–82
 definition of interface, 75–76
 Dependency Inversion principle, 82
 design to interface and, 75
 favoring delegation over inheritance, 80–81
 interface contracts, 76–77
 keeping simple, 79–80
 mock implementations of interfaces, 79
 not for every class, 84
 polymorphism and, 83–84
 separating perspectives in, 77–79
 summary review, 84

Interface-Oriented Design (Pugh), 77

Interfaces

 Commonality-Variability Analysis
 and, 134–136
 coupling and testability of, 166–167
 defining prior to implementing
 code, 13–14
 definition of, 75–76
 Law of Demeter and, 163–165,
 169–171
 laws of, 77
 needs and capabilities, 168–169
 needs vs. capabilities, 167–168
 summary review, 171–172
 users coupled to, 25

Investment vs. debt, refactoring, 156

"Is-a" relationship, UML class diagram,
 192

J

Jacobsen, Ivar, 147–148

Java

 abstract data types, 218
 breaking encapsulation with `get()`,
 63

 Commonality-Variability Analysis
 in, 134–135

 composition vs. aggregation in, 196
 declaring interface in, 76

 encapsulation of object type in, 66

 global variables in, 53–54

 interfaces and abstract classes in, 82

 mixing implementation with inter-
 face in, 84

 new keyword in, 23

Just-in-time design

 avoiding code duplication with, 15

 as delaying extensions to code, 102

 enabling by refactoring to the open-
 closed, 159–161

K

Kolsky, Amir, xxvi

L

Language constructs, polymorphic
 behavior without, 83

Law of Demeter, 164–165, 169

Laws of interfaces, 77

Lazy class, refactoring, 155

Lean-Agile Acceptance Test Driven Development. (Pugh), xxvi

Lean-Agile Pocket Guide for Scrum Teams
 (Shalloway), xxv

Lean principles, xvii–xix

Legacy systems

 degrading code intentionally on,
 102–103

 difficulty of debugging, 10

 refactoring and, 156–157

 writing modifiable code for, 103–107

Loose coupling

 as intentional coupling, 206

 for robust and easy/safe to change
 code, 103

 testability related to, 36–37

M

- Magic numbers, as redundancy, 46
- Magic values, encapsulating with enumerations, 218–219
- Member identity, encapsulation of, 54–56
- Merging process
 - continuous integration and, 119–120
 - development branching and, 110, 112–114
 - merge-back in, 115–117
 - merge cost in test-driven development, 117–119
 - "nightly build" and, 115
 - problem, solution and problem in, 114
 - summary review, 122–123
 - workarounds for CI server bottleneck, 121–122
- Methods
 - cohesion of, 6–7, 204–205
 - as functional steps, 4–5
 - intention-revealing names of, 8
 - Law of Demeter for, 164–165
 - minimizing in interfaces, 79–80
 - as movable, 17–18
 - in testability case study, 37–38
 - unit testing of, 13–15
- Minus sign (-), UML notation, 194
- Mock implementations of interfaces, 79
- Models of programs, creating with UML, 191
- Modifications, code, 15–16
- Movable methods, 17–18
- Moves, refactoring, 11–12
- "Multi-Paradigm Design" (Coplien), xxiv

N

- Naming conventions, intention-revealing, 8

Needs interface

- capabilities vs., 167–168
- defined, 168
- following Law of Demeter for, 169–171
- separating from capabilities interface, 168–169
- summary review, 171–172
- Net Objectives Product Development Series
 - goals of, xvii–xix
 - how to design software in stages, xxi–xxii
 - role of this book in, xix–xx
- .NET, properties, 58
- new keyword, 23–24
- Nightly build process, 115
- Notation for accessibility, UML, 194
- Notes, UML, 196
- Nouns
 - in object-oriented analysis, 127
 - problem with, 130–131
 - warning about using, 127–130

- Null Object Pattern, in debugging, 50
- Number sign (#), UML notation, 194

O

- Object: class notation, UML, 198–199
- Object factory, 67–69
- Object-oriented design. *See also* Open-Closed principle
 - Commonality-Variability Analysis in, 127, 132–133
 - handling variation in, 132–133
 - nouns and verbs in, 127–130
 - perspectives, 133–134
 - polymorphism in, 80
 - real problem of, 130–131
- Object type, encapsulation of, 64–66
- "Once and only once rule" (Beck), 52
- One rule, one place principle, redundancy, 208

Online references

- Buckminster Fuller Institute, xxiii
- inheritance for pluggability and dynamism, 187
- listing patterns by what they encapsulate, 71
- Singleton Pattern, 30

Open-Closed principle

- applying to any change, 151–152
- of coupling, 206
- misuse of inheritance in, 174
- overview of, 147–151
- principles and, 152–153
- refactoring to, 157–159
- for simple interfaces, 79–80

out keyword, 63

Overdesign

- avoiding, 101
- avoiding using Open-Closed principle. *See* Open-Closed principle
- avoiding using refactoring. *See* Refactoring
- causes of, 28

Overloaded constructors, and C++, 31–32

Overloaded operations, abstract data classes, 219–220

override keyword, 66

P

Paired programming, 43

Patterns. *See* Design patterns

Performance, and abstract data classes, 219–220

Perspectives

- in Commonality-Variability Analysis, 133–134
- of creation, 23–24
- overview of, 22–23
- practical considerations, 30
- separating for interface, 77–79
- separating use from construction, 27–30
- of use, 24

Planning

- avoiding over- and under-design when, 99
- up-front testing vs., 40–41

Plus sign (+), UML notation, 194

Policy, encapsulating by, 69

Polymorphism

- avoiding premature hierarchies, 80–81
- in general, 83–84
- interfaces and abstract classes in, 81–82
- open-closed through class, 149–150

Postconditions, interface contracts, 76

Precision, UML for, 192

Preconditions, interface contracts, 76

Prefactoring (Pugh), 12, xxviPrimitives. *See* Encapsulation of primitives

Principles

- of abstract data types, 212
- of cohesion, 204
- of coupling, 206
- defined, 152
- of encapsulation, 209
- of redundancy, 208

Private accessibility, UML notation, 194

Private methods, unit testing behavior, 13–15

Programming by Intention

- advantages of, 6
- debugging and, 10
- demonstration of, 3–5
- development of helper methods in, 147–148
- method cohesion and, 6–7
- modifications and extensions using, 15–16
- movable methods and, 17–18
- overview of, 3
- readability achieved with, 101
- readability and expressiveness using, 7–10

- refactoring and enhancing using, 11–12
- seeing patterns in code and, 16–17
- unit testing and, 13–15
- Properties, .NET, 58
- Protected accessibility, UML notation, 194
- Protocol, interface contract, 76
- Public accessibility, UML notation, 194
- public keyword, 53–54
- Public methods, interfaces as sets of, 76
- Pugh, Ken, xxvi

Q

Quality, code. *See* Code qualities

R

- Readability
 - as code quality, 101
 - Programming by Intention and, 7–10
- Redundancy
 - Christmas-tree light analogy of, 203
 - as code quality, 100, 207–208
 - design patterns reducing, 48
 - encapsulation and, 209
 - redefining, 46–47
 - Shalloway's principle avoiding, 45
 - specialization branching and, 110–111
 - testability related to, 36–37
 - types of, 46
- ref keyword, 63
- Refactoring
 - debt vs. investment and, 155–156
 - definition of, 154
 - just-in-time design using, 159–161
 - legacy systems and, 156–157
 - for new requirements, 102
 - to the open-closed, 157–159
 - overview of, 154–155

- Programming by Intention and, 11–12
- reasons for, 155
- redundancies, 208
- unit testing and, 14
- Refactoring: Improving the Design of Existing Code* (Fowler)
 - about author, xxiv
 - cleaning up messy/poor code, 11, 102
 - on refactoring moves, 154–155
- Reference objects
 - breaking encapsulation with `get()`, 62–64
 - encapsulation of, 59–61
- Relationships, in UML
 - class diagrams showing, 192–195
 - "has-a" relationship showing, 195–196
 - indicating number of things of other object, 197
- `releaseInstance()` method, 32
- Representational coupling, 206
- Rework
 - avoiding in merge-back process, 116
 - minimizing complexity and, 99, 102
- Robocode example, 152–153, 157–159
- Runtime, inheritance vs. delegation at, 180–181

S

- Scalability, inheritance and, 183–184
- Scope, ATDD improving clarity of, 42
- Self-encapsulating data members
 - overview of, 56–57
 - preventing changes, 58–59
- Separation of concerns
 - in cohesion, 204
 - in delegation vs. inheritance, 179
 - and inheritance, 177
- Separation of Use from Construction, 21
- Sequence diagrams, UML, 198–199

Serialization, separating use from construction, 28

Servers

- continuous integration of, 121–122
- coupling and testability of, 166–167
- Law of Demeter and, 169–171
- separating needs/capabilities interfaces, 168–169
- summary review, 171–172

Service-Oriented Architecture, 27

`Service_Impl` class, 23–24

Services

- coupling and testability of, 166–167
- Law of Demeter and, 169–171
- needs vs. capabilities and, 167–168
- perspectives and, 22–24
- separating use from construction, 27–30
- what you hide you can change and, 25–27

`set()` method

- breaking encapsulation with, 62–64
- difficulty of encapsulating reference objects, 59–61
- encapsulating data members behind, 56–57
- preventing changes, 58–59

`setter()` methods, 28

`setX()` method, preventing changes, 58

Shalloway, Alan, 71–72, xxv

Shalloway's law and Shalloway's principle

- corollaries to, 51
- defining, 45
- design patterns reducing redundancy, 48
- redundancy and other code qualities, 50–52
- refining redundancy, 46–47
- time spent finding bugs, 48–50
- as trim tab, xxii

types of redundancy, 46–48

Shotgun surgery, refactoring, 155

Single Responsibility principle

- cohesion of class and, 204
- defined, 17

Singleton pattern, 30

Source code branching

- development branching, 112–114
- merge-back, 115–117
- overview of, 109–110
- problem, solution, problem, 114
- specialization branching, 110–111
- version control creating, 109

Specialization branching, 109–111

Specification perspective

- abstract classes and, 82
- in interface-oriented design, 75
- separating from implementation perspective, 77–79, 84

Static methods, 30

Static relationships, showing in class diagrams, 198

Strategy Pattern

- delegation vs. inheritance and, 179
- encapsulating single varying algorithm with, 70–71
- encapsulation of type in, 67–68
- inheritance and, 183
- scalability of, 183–184
- seeing patterns in, 17
- testing issues, 185–186

Strings, and abstract data types, 214

Strong cohesion, 204

Subclass coupling, 206

Switch creep, 39

T

TDADD (Test Driven Analysis and Design and Development), 40

TDD. *See* Test-Driven Development (TDD)

- Team Foundation Server, 121
 - TeamCity, 121
 - Template Method Pattern
 - removing redundant implementations with, 48
 - seeing patterns in code, 16
 - and Strategy Pattern, 17
 - Test Driven Analysis and Design and Development (TDADD), 40
 - Test-Driven Development (TDD)
 - better design, 42
 - defining testing, 35–36
 - improving clarity of scope, 42
 - and merge cost, 117–119, 123
 - other advantages, 43
 - overview of, 35
 - paired programming and, 43
 - programmers as frogs and, 39
 - reducing complexity with, 42
 - testability and code quality, 36–37
 - testability case study, 37–38
 - testing interface to clarify contract, 76–77
 - up-front testing vs., 39–41
 - Testing and testability. *See also* Unit testing; Up-front testing
 - clarifying interface contract through, 76–77
 - cohesion, 205
 - coupling, 166–167, 207
 - encapsulation, 210
 - redundancy, 51, 208
 - simplifying in ADTs by narrowing of contract, 213
 - use of inheritance in, 185–186
 - Text, using as external values, 215–217
 - Textual commands, interface as set of, 76
 - Tight coupling, 206
 - Time between merges, development
 - branching
 - merge-backs and, 116–117
 - overview of, 113–114
 - test-driven development and merge cost, 117–119
 - The Timeless Way of Building* (Alexander), 40, xxiv
 - Trim tabs. *See also* Testing and testability, xxi–xxii
 - Trunk
 - in continuous integration, 119–120
 - in development branching, 110, 112
 - in merge-back process, 115–117
 - in specialization branching, 109–111
 - Types, creators coupled to, 25
- ## U
- UML Distilled* (Fowler), 200, xxiv
 - Under-design, avoiding, 101
 - Unencapsulated code, global variables and, 53–54
 - Unified Modeling Language (UML)
 - class diagrams and, 192–193
 - class diagrams showing relationships, 194–195
 - composition and uses, 195–196
 - composition vs. aggregation, 196
 - definition of, 191
 - indicating number of things another object has, 197
 - notation for accessibility, 194
 - notes in, 196
 - object: class notation, 198–199
 - polymorphism in, 83
 - reasons to use, 192
 - sequence diagram, 198
 - showing dependence using dashes, 198
 - showing "has-a" relationship, 195
 - summary review, 200
 - Unit Test-Driven Development (UTDD), 41–42
 - Unit testing
 - creating automated acceptance tests with, 39
 - creating from acceptance tests, 40, 44

Unit testing (*continued*)
 inserting mock test into interface
 for, 79
 in Programming by Intention, 13–15
 Unit Test-Driven Development, 41–42
 Up-front testing. *See also* Testing and
 testability
 of cohesion, 205
 of coupling, 207
 no excuses for avoiding, 43
 overview of, 35
 of redundancy, 208
 reflection on, 39–41
 Use, separating construction from
 approach to, 27–30
 coupling from perspective of, 25–27
 debugging and, 48–50
 perspective of, 24
 question to ask, 21–22
 realistic approach to, 27–30
 timing your decisions, 30–31
 validating concept for yourself,
 32–33
 "Uses-a" relationship, UML class dia-
 gram, 192

V

Validation
 Acceptance Test-Driven Develop-
 ment and customer, 95–96

defining requirements using tests
 for, 87
 of separation of use from construc-
 tion, 32–33
 using text as external values, 216
 Values
 breaking encapsulation with `get()`,
 63
 encapsulating magic, 218–219
 using text as external, 215–217
 Variability Analysis, 133
 Variations. *See also* Commonality-
 Variability Analysis
 encapsulating all, 209
 handling in problem domain, 132
 hiding with inheritance, 179,
 181–182
 in Variability Analysis, 133
 Verbs
 in object-oriented analysis, 127
 problem with, 130–131
 warning about using, 127–130
 Version control. *See also* Continuous
 integration (CI), 109
 Viscosity, avoiding code, 103

W

Weak cohesion, 204
 Web Services Description Language
 (WSDL), 76