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

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

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | QUE | PRENTICE HALL | SAMS | Safari Books Online

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

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

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
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.

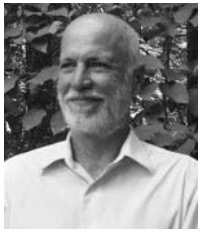
There would be no Net Objectives without our customers. We love our customers.

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, sustainable 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

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W