Extracted from:

Interface-Oriented Design

Contents

Preface

Interface-Oriented Design explores how you can develop software with interfaces that interact with each other. We'll look at techniques for breaking down solutions into these interacting interfaces and then for determining appropriate implementations for these interfaces to create well-structured programs. We have plenty of examples that will show you ways to create effective designs composed of interfaces to objects, components, and services. And we'll even have some fun along the way.

You've probably learned about (and experienced) software development using object-oriented design. Interface-oriented design concentrates on the interfaces of modules, which may or may not be implemented with object-oriented languages. Designs that emphasize interfaces are loosely coupled—and that's a good thing. If you have only an interface to which to code, you cannot write code dependent on an implementation, which helps keep us honest.

Distributed computing, such as service-oriented architectures, places a particular emphasis on interfaces. The interfaces may be procedure oriented (such as Remote Procedure Calls) or document oriented (such as web services). We'll explore the transparency and loose coupling traits that are key to distributed interfaces to help you build better distributed systems.

Inheritance is often a tricky technique to get correct—it is often one of the most abused features in object-oriented languages. We'll look at designs that employ inheritance versus ones that emphasize interfaces to demonstrate the trade-offs between the two.

This ongoing emphasis on interfaces may seem a bit extreme. But by looking at one extreme, you'll start to see a different viewpoint that can give you fresh insights into your current approach to software development.

Here then is a road map for our journey through interface-oriented design.

Road Map

Chapter 1, *Introduction to Interfaces*

We'll start by ordering pizza. One should never read a book on an empty stomach, so we'll use the activities of finding a suitable pizza shop and ordering a pizza as a nonprogramming introduction to interfaces. We'll then briefly look at some code and textual interfaces as introductory background for topics we'll explore in later chapters.

Chapter 2, *Interface Contracts*

It's hard to use an interface if an implementation provides no guarantee of working successfully. We'll see how the Three Laws of Interfaces applies to implementations and how Design by Contract helps in understanding an interface's protocol. Finally, you'll need to test an implementation to verify that it lives up to its side of the contract.

Chapter 3, *Interface Ingredients*

You can structure interfaces in many ways, including pull versus push and stateful versus stateless interfaces. We'll explore the trade-offs and benefits of these facets and finish by outlining how to transform an interface from one facet to another.

Chapter 4, *What Should Be in an Interface?*

An interface should have cohesive functionality. There are no absolute rules to what makes a cohesive interface, but we'll look at different sets of interfaces to explore the concept of cohesiveness and see how it helps development.

Chapter 5, *Inheritance and Interfaces*

Inheritance in object-oriented programs is often overused. We'll investigate better ways to organize designs using interfaces and delegation and discover the trade-offs and benefits over inheritance.

Chapter 6, *Remote Interfaces*

Many programs these days are dependent on communicating with remote interfaces. We'll look at the ramifications of using remote

interfaces, see why document-style interfaces are becoming more common, and learn how to best organize one.

Chapter 7, *A Little Process*
: Interface-oriented design is but one part of the overall development process. We'll see how it fits in, and we'll get ready for the three design examples in the following chapters.

Chapter 8, *Link Checker*
: In this chapter, we'll develop a miniproject: a link checker for web pages, demonstrating how interfaces provide flexibility in selecting implementations.

Chapter 9, *Web Conglomerator*
: Why rely on web sites to put information together in the way that you want it? The web conglomerator project gathers information into a single page and lets us explore interface cohesiveness and interface generalization as we create this program.

Chapter 10, *Service Registry*
: Remote services use directories to help you locate a service provider. In this project, we'll develop a service registry to explore how directory services work and see an example of a document-style interface.

Chapter 11, *Patterns*
: The well-known "Gang of Four" book divides patterns into two camps: class-based and object-based. To give another viewpoint, we'll look at some of those classic patterns as being interface-based instead.

Who Should Read This Book

This book is aimed at developers who have some experience with programming and who have been exposed to object-oriented design. Even if you are heavy into object orientation, you might find the interface-oriented approach helps you gain some insight into different ways of approaching a design. Understanding interfaces will help you transition to designing Service-Oriented Architectures.

About the Cover

Concentrating on interfaces is key to decoupling your modules.[1] You probably learned to type on a QWERTY keyboard, as shown on the cover. That interface is the same regardless of whether the implementation is an old-fashioned typewriter, a modern electric typewriter, or a computer keyboard. There have been additions to the keyboard, such as function keys, but the existing layout continues to persist.

But other layouts, such as Dvorak,[2] are more efficient for typing. You can switch your computer keyboard to use an alternate layout; the switching module works as an adapter. Inside the keyboard driver, the keystrokes are converted to the same characters and modifiers (e.g., Shift, Alt, etc.) that are produced by the regular keyboard.

The QWERTY keyboard layout was derived from concern about implementation. According to one web site,[3] "It is sometimes said that it was designed to slow down the typist, but this is wrong; it was designed to allow *faster* typing—under a constraint now long obsolete. In early typewriters, fast typing using nearby type-bars jammed the mechanism. So Sholes fiddled the layout to separate the letters of many common digraphs (he did a far from perfect job, though; *th*, *tr*, *ed*, and *er*, for example, each use two nearby keys). Also, putting the letters of *typewriter* on one line allowed it to be typed with particular speed and accuracy for demos. The jamming problem was essentially solved soon afterward by a suitable use of springs, but the keyboard layout lives on."

Creating interfaces that are easy to use and decoupling their use from their implementation are two facets that we'll explore a lot in this book. (And you may have thought the cover was just a pretty picture.)

So, What Else Is in Here?

Simple Unified Modeling Language (UML) diagrams show the class and interface organization throughout the book. We use Interface-Responsibility-Interaction (IRI) cards, a variation of Class-Responsibility-Colla-

[1] As Clemens Szyperski puts it, "The more abstract the class, the stronger the decoupling achieved." See http://www.sdmagazine.com/documents/sdm0010k/.

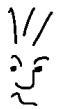
[2] See http://www.microsoft.com/enable/products/dvlayout.aspx.

[3] See http://www.ctrl-c.liu.se/~ingvar/jargon/q.html.

boration (CRC) cards, as the primary method for creating an interface-oriented designs. You'll also find code examples in multiple languages to show how interfaces are implemented in those languages.

On a terminology note, the OMG Modeling Language Specification (revision 1.3) uses the phrase *realize interface*, which means a component implements the services defined in the interface. Allen Holub in *Holub on Patterns* uses the term *reify*, which means "consider an abstract concept to be real." I thought about alternating one of those verbs with the word *implements*, but they are less familiar. If you get tired of seeing *implementing*, just imagine it's *reify*.

You will see a few sections that look like this:



"Joe Asks..."
These sections provide answers for some common questions.

Acknowledgments

I would like to thank my reviewers for reading the draft copies of this book and contributing numerous comments that helped improve the book. Thanks to David Bock, Tom Ball, Ron Thompson, Gary K. Evans, Keith Ray, Rob Walsh, David Rasch, Carl Manaster, Eldon Alameda, Elias Rangel, Frédérick Ros, J. Hopkins, Mike Stok, Pat Eyler, Scott Splavec, Shaun Szot, and one anonymous reviewer. Thanks to Michael Hunter, an extraordinary tester, who found a lot of "bugs" in this book. Thanks to Christian Gross, a reviewer who gave me many suggestions that just couldn't fit into this book and to Kim Wimpsett for proofreading the manuscript.

I appreciate Andy Hunt, my editor and publisher, for encouraging me to write this book, and all his help with the manuscript.

Thanks also to Leslie Killeen, my wife, for putting up with me writing another book just as soon as I finished my previous book, *Prefactoring*, winner of the 2006 Software Development Jolt Product Excellence Award.[4]

And now, here we go!

[4] See http://www.ddj.com/dept/architect/187900423?pgno=3/.