



# Elemental Design Patterns

J A S O N   M c C .   S M I T H

*Foreword by* **Grady Booch**

# Elemental Design Patterns

*This page intentionally left blank*

# Elemental Design Patterns

Jason McC. Smith

◆ 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 author 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](http://informit.com/aw)

*Library of Congress Cataloging-in-Publication Data*

Smith, Jason McC. (Jason McColm)  
Elemental design patterns / Jason McC. Smith.  
p. cm.

Includes bibliographical references and index.

ISBN 0-321-71192-0 (hardcover : alk. paper) 1. Software patterns. 2. Software architecture  
3. System design. I. Title.  
QA76.76.P37S657 2012  
005.1—dc23

2012001271

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: 978-0-321-71192-2

ISBN 10: 0-321-71192-0

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.  
First printing, March, 2012

*For B.*

*You were there at the beginning of this journey,  
I wish you'd been able to see the end.*

*This page intentionally left blank*

# Contents

*Figures*    *xi*

*Tables*    *xv*

*Listings*    *xvii*

*Foreword*    *xix*

*Preface*    *xxi*

*Acknowledgments*    *xxiii*

*About the Author*    *xxv*

## **1 Introduction to Design Patterns    1**

- 1.1 Tribal Musings    5
- 1.2 Art or Science?    9
  - 1.2.1 Viewing Patterns as Rote    9
  - 1.2.2 Language-Dependent Views    10
  - 1.2.3 From Myth to Science    12

## **2 Elemental Design Patterns    13**

- 2.1 Background    14
- 2.2 The Where, the Why, the How    17
  - 2.2.1 Decomposition of *Decorator*    18
  - 2.2.2 Down the Rabbit Hole    21
  - 2.2.3 Context    30
  - 2.2.4 The Design Space    33
- 2.3 Core EDPs    42
- 2.4 Conclusion    44

## **3 Pattern Instance Notation    45**

- 3.1 Basics    45
- 3.2 The PINbox    49
  - 3.2.1 Collapsed PINbox    49



3.2.2	Standard PINbox	51
3.2.3	Expanded PINbox	55
3.2.4	Stacked PINboxes and Multiplicity	56
3.2.5	Peeling and Coalescing	62
3.3	Conclusion	65
<b>4</b>	<b>Working with EDPs</b>	<b>67</b>
4.1	Composition of Patterns	68
4.1.1	Isotopes	72
4.2	Recreating <i>Decorator</i>	77
4.3	Refactoring	91
4.4	The Big Picture	101
4.5	Why You May Want to Read the Appendix	105
4.6	Advanced Topics	108
4.6.1	Focused Documentation and Training	108
4.6.2	Metrics	109
4.6.3	Procedural Analysis	112
4.7	Conclusion	112
<b>5</b>	<b>EDP Catalog</b>	<b>115</b>
Create Object		117
Retrieve		126
Inheritance		130
Abstract Interface		140
Delegation		145
Redirection		151
Conglomeration		159
Recursion		165
Revert Method		172
Extend Method		181
Delegated Conglomeration		187
Redirected Recursion		193
Trusted Delegation		200
Trusted Redirection		209
Deputized Delegation		216
Deputized Redirection		222
<b>6</b>	<b>Intermediate Pattern Compositions</b>	<b>229</b>
Fulfill Method		231
Retrieve New		235

Retrieve Shared	240
Objectifier	244
Object Recursion	251

## **7 Gang of Four Pattern Compositions 259**

7.1 Creational Patterns	260
7.1.1 Abstract Factory	260
7.1.2 Factory Method	263
7.2 Structural Patterns	265
7.2.1 Decorator	265
7.2.2 Proxy	269
7.3 Behavioral Patterns	273
7.3.1 Chain of Responsibility	273
7.3.2 Template Method	275
7.4 Conclusion	279

## **A $\rho$ -Calculus 281**

A.1 Reliance Operators	282
A.2 Transitivity and Isotopes	285
A.3 Similarity	286
A.4 EDP Formalisms	287
A.5 Composition and Reduction Rules	291
A.6 Pattern Instance Notation and Roles	293
A.7 EDP Definitions	295
A.7.1 Create Object	295
A.7.2 Retrieve	296
A.7.3 Inheritance	298
A.7.4 Abstract Interface	298
A.7.5 Delegation	299
A.7.6 Redirection	300
A.7.7 Conglomeration	300
A.7.8 Recursion	301
A.7.9 Revert Method	301
A.7.10 Extend Method	302
A.7.11 Delegated Conglomeration	303
A.7.12 Redirected Recursion	303
A.7.13 Trusted Delegation	304
A.7.14 Trusted Redirection	305
A.7.15 Deputized Delegation	306
A.7.16 Deputized Redirection	307

A.8 Intermediate Pattern Definitions	308
A.8.1 Fulfill Method	308
A.8.2 Retrieve New	309
A.8.3 Retrieve Shared	310
A.8.4 Objectifier	311
A.8.5 Object Recursion	312
A.9 Gang of Four Pattern Definitions	313
A.9.1 Abstract Factory	313
A.9.2 Factory Method	314
A.9.3 Decorator	316
A.9.4 Proxy	317
A.9.5 Chain of Responsibility	318
A.9.6 Template Method	319
<b>Bibliography</b>	<b>321</b>
<b>Index</b>	<b>325</b>

# Figures

2.1	<i>Decorator's</i> usual example UML. . . . .	19
2.2	<i>Objectifier</i> as UML. . . . .	20
2.3	<i>Object Recursion</i> as UML. . . . .	20
2.4	A simple method call as UML. . . . .	23
2.5	The parts of a method call. . . . .	31
2.6	A simple design space. . . . .	34
2.7	A simple design space with EDPs. . . . .	35
2.8	Our first four EDPs. . . . .	39
2.9	The design space extended to three dimensions. . . . .	39
2.10	The design space with method similarity fixed to similar. . . . .	40
2.11	<i>Recursion</i> Example UML. . . . .	42
2.12	<i>Deputized Redirection</i> example UML. . . . .	42
3.1	UML collaboration diagram. . . . .	47
3.2	<i>Strategy</i> as pattern:role tags in UML. . . . .	48
3.3	Huge UML of a not-so-huge system. . . . .	48
3.4	Multiple instances of <i>Strategy</i> as pattern:role tags in UML. . . . .	49
3.5	Collapsed PINbox. . . . .	50
3.6	Collapsed PINbox as annotation. . . . .	50
3.7	<i>Singleton</i> and <i>Abstract Factory</i> in class diagram. . . . .	50
3.8	<i>Template Method</i> in sequence diagram. . . . .	51
3.9	Standard PINbox. . . . .	51
3.10	PIN used with UML class diagram. . . . .	52
3.11	PIN used with UML sequence diagram. . . . .	53
3.12	Standard PIN role connections. . . . .	54
3.13	Blank expanded PIN instance. . . . .	55
3.14	Expanded PIN instance. . . . .	56
3.15	Expanded PIN instance using UML. . . . .	57
3.16	A need for multiple related PINboxes. . . . .	59

3.17	Stacked PINbox. . . . .	60
3.18	Multiple <i>Strategy</i> instances as PINboxes. . . . .	61
3.19	Showing the interaction between multiple <i>Strategy</i> PINboxes. . . . .	62
3.20	<i>Abstract Factory</i> as part of a larger UML diagram. . . . .	63
3.21	<i>Abstract Factory</i> subsumed within the expanded PINbox. . . . .	64
3.22	Coalesced PINbox. . . . .	65
4.1	<i>Abstract Interface</i> and <i>Inheritance</i> EDPs as UML. . . . .	68
4.2	Internal definition of <i>Fulfill Method</i> as UML. . . . .	69
4.3	<i>Fulfill Method</i> as simple connected PINboxes. . . . .	69
4.4	<i>Fulfill Method</i> as expanded PINbox. . . . .	69
4.5	<i>Fulfill Method</i> as standard PINbox. . . . .	70
4.6	Flipping our EDPs in <i>Fulfill Method</i> —oops. . . . .	71
4.7	Flipped EDPs as PINboxes. . . . .	72
4.8	Alternative classes that can fulfill an <i>Abstract Interface</i> EDP. . . . .	75
4.9	Alternative structures that can fulfill an <i>Inheritance</i> EDP. . . . .	76
4.10	<i>Decorator</i> 's usual example UML. . . . .	78
4.11	<i>Fulfill Method</i> definition as annotated UML. . . . .	79
4.12	<i>Objectifier</i> UML annotated with PIN. . . . .	80
4.13	<i>Objectifier</i> and <i>Trusted Redirection</i> . . . . .	81
4.14	<i>Object Recursion</i> annotated with PIN. . . . .	82
4.15	<i>Object Recursion</i> as just PIN. . . . .	83
4.16	<i>Object Recursion</i> and <i>Extend Method</i> . . . . .	84
4.17	<i>Decorator</i> annotated with PIN. . . . .	85
4.18	<i>Decorator</i> as PIN. . . . .	86
4.19	<i>Decorator</i> instance as a PINbox. . . . .	86
4.20	Expanding <i>Decorator</i> : one level. . . . .	87
4.21	Expanding <i>Decorator</i> : two levels. . . . .	88
4.22	Expanding <i>Decorator</i> : three levels. . . . .	89
4.23	Expanding <i>Decorator</i> : four levels. . . . .	90
4.24	<i>Delegation</i> before <i>Rename Method</i> refactoring. . . . .	93
4.25	<i>Delegation</i> after <i>Rename Method</i> refactoring— <i>Redirection</i> . . . . .	94
4.26	<i>Delegation</i> before <i>Move Method</i> refactoring. . . . .	95
4.27	The design space with method similarity fixed to dissimilar. . . . .	96
4.28	<i>Delegation</i> after <i>Move Method</i> refactoring: boring case. . . . .	97
4.29	<i>Delegation</i> after <i>Move Method</i> refactoring: into same type. . . . .	97
4.30	<i>Delegation</i> after <i>Move Method</i> refactoring: <i>Delegated Conglomeration</i> . . . . .	97
4.31	<i>Delegation</i> after <i>Move Method</i> refactoring: <i>Conglomeration</i> . . . . .	98
4.32	<i>Delegation</i> after <i>Move Method</i> refactoring: <i>Trusted Delegation</i> . . . . .	98

4.33	<i>Delegation</i> after <i>Move Method</i> refactoring: <i>Revert Method</i> .	99
4.34	<i>Delegation</i> after <i>Move Method</i> refactoring: <i>Deputized Delegation</i> .	99
4.35	Summarizing refactoring effects so far.	100
4.36	Implicit used-by relationships among the EDPs and selected other patterns.	102
4.37	The full method-call EDP design space: dissimilar method.	103
4.38	The full method-call EDP design space: similar method.	104
4.39	Method-call EDP refactoring relations.	106
5.1	Polymorphic approach	173
5.2	Subclassing approach	175
5.3	UI class cluster showing an instance of <i>Trusted Delegation</i> .	203
5.4	UI class cluster showing an instance of <i>Trusted Redirection</i> .	211
5.5	UI class cluster showing an instance of <i>Deputized Delegation</i> .	218
5.6	UI class cluster showing an instance of <i>Deputized Redirection</i> .	225
7.1	<i>Abstract Factory</i> subsumed within the expanded PINbox.	261
7.2	Reducing the diagram to just one instance of <i>Abstract Factory</i> .	262
7.3	Simplifying Figure 7.2.	264
7.4	<i>Abstract Factory</i> as PIN only.	265
7.5	<i>Factory Method</i> subsumed within the expanded PINbox.	266
7.6	<i>Factory Method</i> as PIN only.	267
7.7	<i>Decorator</i> subsumed with the expanded PINbox.	268
7.8	<i>Decorator</i> as PIN only.	269
7.9	<i>Decorator</i> expanded three levels deep and flattened.	270
7.10	<i>Proxy</i> subsumed with the expanded PINbox.	271
7.11	<i>Proxy</i> as PIN only.	272
7.12	<i>Proxy</i> PIN reorganized to better match <i>Decorator</i> .	272
7.13	<i>Chain of Responsibility</i> subsumed within the expanded PINbox.	274
7.14	<i>Chain of Responsibility</i> as PIN only.	275
7.15	<i>Template Method</i> subsumed within the expanded PINbox.	276
7.16	<i>Template Method</i> reduced to a single instance.	277
7.17	<i>Template Method</i> as PIN only.	278
7.18	<i>Template Method</i> PIN reorganized to better match <i>Decorator</i> .	279
7.19	<i>Factory Method</i> redefined using <i>Template Method</i> .	279
A.1	The full method call EDP design space: similar method	288
A.2	The full method call EDP design space: dissimilar method	289
A.3	Standard PINbox	294
A.4	Expanded PIN instance	294

*This page intentionally left blank*

# Tables

2.1	Pattern pieces sorted into three categories of a pattern definition . . . .	22
2.2	All interactions between entities of object-oriented programming . . . .	28
2.3	Nonscoping interactions between entities of object-oriented programming . . . . .	28
A.1	All interactions between entities of object-oriented programming . . . .	283
A.2	Nonscoping interactions between entities of object-oriented programming . . . . .	284



*This page intentionally left blank*

# Listings

2.1	A simple method call as pseudocode. . . . .	24
2.2	Fields within classes, instances, and namespaces, as defined and used in C++. . . . .	26
2.3	A Java class, and one possible equivalent object and type. . . . .	27
2.4	Typing as context. . . . .	29
2.5	A method call chain as pseudocode. . . . .	30
2.6	Simple method call for Figure 2.5. . . . .	31
2.7	Example of a <i>Recursion</i> method call in Java. . . . .	36
2.8	Example of a <i>Delegation</i> method call in C++. . . . .	36
2.9	Example of a <i>Redirection</i> method call in Objective-C. . . . .	37
2.10	Example of a <i>Conglomeration</i> method call in Java. . . . .	38
5.1	Uninitialized data. . . . .	118
5.2	Fixed default values. . . . .	120
5.3	Dynamic initialization . . . . .	121
5.4	<i>Create Object</i> Implementation. . . . .	125
5.5	<i>Retrieve</i> with an update. . . . .	126
5.6	<i>Retrieve</i> in a temporary variable. . . . .	127
5.7	Basic inheritance example in Objective-C. . . . .	131
5.8	Overriding an implementation. . . . .	132
5.9	Implementation assumption mismatch. . . . .	133
5.10	Obvious fix—but likely not feasible. . . . .	134
5.11	Fixing a bug while leaving old code in place. . . . .	134
5.12	Using <i>Redirection</i> to hide part of an interface. . . . .	137
5.13	Animals almost all move but in very different ways. . . . .	141
5.14	CEO delegates out responsibilities. . . . .	145
5.15	Tom paints the fence with help. . . . .	152
5.16	Prep work and cleanup are important. . . . .	153
5.17	Prep work and cleanup are decomposable. . . . .	160

5.18	Instance swapping for protocol fallback in C++.	173
5.19	Auto fallback/forward using <i>Revert Method</i> .	176
5.20	Using <i>Redirection</i> in Python to add behavior.	182
5.21	Using <i>Extend Method</i> to add behavior.	182
5.22	Inviting friends naively in Java.	187
5.23	A slightly better approach for inviting friends.	188
5.24	<i>Delegated Conglomeration</i> in Java.	188
5.25	Traditional iteration and invocation in C.	193
5.26	Object-oriented iteration and invocation in C++.	193
5.27	Basic <i>Redirected Recursion</i> in C++.	194
5.28	Paratroopers implementing <i>Redirected Recursion</i> .	195
5.29	UI widgets demonstrating <i>Trusted Delegation</i> in C++.	201
5.30	Event handler in C++ showing <i>Trusted Redirection</i> .	210
5.31	UI widgets demonstrating <i>Deputized Delegation</i> in C++.	216
5.32	UI widgets demonstrating <i>Deputized Redirection</i> in C++.	222
6.1	Conditionals to select behavior.	246
6.2	Using <i>Objectifier</i> to select behavior.	247
A.1	Simple code example	287

# Foreword

There's a wonderful scene in the movie *2001: A Space Odyssey* that comes to mind.

Having spent several months alone on the derelict ship *Discovery*—and that after having earlier lobotomized the errant Hal—Dr. David Bowman approaches a monolith that draws him in to a new world. His final message back to earth ends “It’s full of stars!”

Software-intensive systems are new worlds that we create with our own mental labor. Whereas the world that Bowman saw was formed from atoms and thus full of stars, our worlds are formed from bits...and are full of patterns.

Whether intentional or not, all well-structured, software-intensive systems are full of patterns. Identifying the patterns in a system serves to raise the level of abstraction in reasoning about that system; imposing patterns on a system serves to bring even further order, elegance, and simplicity to that system. In my experience, patterns are one of the most important developments in software engineering in the past two decades.

I’ve had the pleasure of working with Jason as he evolved his work on SPQR, and let me assure you that he has contributed greatly to the advance of the understanding and practice of patterns. *Elemental Design Patterns* will help you think about patterns in a new way, a way that will help you apply patterns to improve the software worlds that you create and evolve. If you are new to patterns, this is a great book to start your journey; if you are an old hand with patterns, then I expect you’ll learn some new things. I certainly did.

Grady Booch  
IBM Fellow  
February, 2012

*This page intentionally left blank*

# Preface

This book is an introduction to a new class of design pattern, the Elemental Design Patterns, which form a foundation for the study and application of software engineering design patterns. Its foundations are in research into the very fabric of software programming theory, but it is intended to be practical and pragmatic. It is intended for both the beginning programmer and the seasoned developer. It should help students engage with the software industry and give researchers new points to ponder.

In short, this book is meant to be *used*.

By the end of it, you should have a new set of tools in your toolbox, a richer understanding of some of the basic concepts of programming that we all use every day, and knowledge of how they relate and interact with one another to do amazing things. The Elemental Design Patterns, or EDPs, are a collection of fundamental programming ideas that we use reflexively and probably don't think twice about when doing so. This body of work gives them explicit descriptions, regularized names to use in discussions, and a framework for using them in concert and for comparing them on their own merits. If you're a new student, you'll learn that instead of facing the ever-growing design patterns literature as a collection of daunting all-or-nothing blocks, you have a chance to take them on piece by piece and gradually understand the literature in a methodical way. If you're an old hand at software design and patterns, you'll find new ways to look at old approaches and see new opportunities for our discipline.

This book assumes you have a passing familiarity with design patterns as a field but have not used or studied them in detail. Knowing that they exist and having a brief colloquial knowledge of what they are is enough to start the discussion. The book does not assume you have a background in programming theory, language design, or even a strong one in object-oriented programming, just a desire to learn how to think critically about software design. These subjects will be touched on but only as a starting point for those interested in diving deeper into them through

the provided references. The Unified Modeling Language is used to describe small examples, and I suggest either [20] or [33] as references if you do not already know UML. You should have a basic foundation in programming, either procedural or object oriented. The latter will help, but it's not absolutely required—this text provides much of the necessary information to explain object-oriented programming in easily digestible chunks. Developers experienced with object-oriented systems may still be surprised at finding new perspectives on concepts that they thought they had mastered long ago and a greater appreciation for object-oriented programming as a whole.

Many programmers see the “design patterns community” as an esoteric body of experts and one that they themselves are not a part of. By giving you a new perspective on what can constitute a design pattern, this book should convince you that *every* programmer is a member of the design patterns community, whether they know it or not. Every single programmer uses design patterns every time they write a line of code, even if don't think of it that way. Nor are they likely to realize the options they have at their disposal. Design patterns are the shared conceptual space in which we write the electronic dreams that shape our world. It's time we had a map of the landscape in which we work and play.

Following the example of the seminal Gang of Four text [21], this book is divided into two sections. First is a discussion of why this book was written and who it is written for and an explanation of what EDPs are, where they came from, and why they're important. This section explains the rationale, the *why*, behind the EDPs. Next is an introduction to the Pattern Instance Notation, a diagramming system for working with patterns at many levels of granularity and in a multitude of environments. Wrapping up this first section is a discussion of how EDPs can be used to build up to, and in conjunction with, the greater design patterns literature. The second section of the book is a collection of design patterns, starting with the EDPs and working through examples of how they combine to form Intermediate patterns, and finally, a selection of the Gang of Four patterns recast as EDP compositions. The EDPs presented here are only a portion of the EDP Catalog, a collection of the first round of defined and described fundamental patterns. The software engineering community will continue to define and refine additional EDPs as the underlying concepts take root. I hope you decide to help in the endeavor.

Welcome, it's good to have you join us.

# Acknowledgments

I have many people to thank for this book coming to life. In not quite chronological order. . .

From the University of North Carolina at Chapel Hill, David Stotts, my Ph.D. advisor who oversaw the birth of SPQR and the EDPs over many years; also my committee, who, even though they were convinced it was probably infeasible, thought it would be an interesting journey and let me go for the brass ring anyway: Jan Prins, David Plaisted, Al Segars, and Sid Chatterjee. You each added invaluable help at critical times.

From my years at IBM Watson Research in New York, Sid Chatterjee again, who convinced me to come play in the Big Blue Playpen; Clay Williams, who gave me free rein to pursue these crazy ideas further and with whom I still miss having coffee; Peter Santhanam, who championed those ideas and from whom I learned a greater appreciation for legacy systems; Brent Hailpern, from whom I learned many valuable lessons in management, the dark humor of corporate life and simple humanity; Edith Schonberg, who put up with my shenanigans more than any manager should have to; and many others who listened to me maniacally talk about this body of work at every turn. My friends, I miss you all.

Also from IBM but deserving a special mention, Grady Booch, who took me under his wing for a wild ride that I wouldn't have traded for anything. Grady, your guidance, mentoring, and advocacy have been immeasurable, and I look forward to future collaborations and continued friendship.

From The Software Revolution, Inc., in Kirkland, Washington, where I am now Senior Computer Scientist, I have to thank everyone for being understanding and supportive of my need to commit this information to paper. It has been a true pleasure working with all of you, and I am eager to see where we can take our company.

To my many reviewers, your advice and comments were highly insightful and helpful. You made this book a much better product, and you have my deepest thanks: Lee Ackerman, Lars Bishop, Robert Bogetti, Robert Couch,



Bernard Farrell, Mary Lou Hines Fritts, Gail Murphy, Jeffrey Overbey, Ethan Roberts, Carlota Sage, Davie Sweis, Peri Tarr, and Rebecca Wirfs-Brock. Elizabeth Ryan, Raina Chrobak, Chris Zahn, and Chris Guzikowski at Addison-Wesley were the model of compassionate support during the trials of this process—my thanks to you and the rest of the crew there, with a special thanks to Carol Lallier, whose expert polish on this book was invaluable.

On a personal note, I thank my friends and family, who have been incredibly patient while I have put in seemingly endless hours on this, even though they were hoping to see more of me now that I've moved back to the Seattle area.

Finally, my wife Leah. You have supported me in so many large and small ways throughout our time together. You have given your time, your patience, and your love, and you have my immense love and gratitude. Thank you. Words are simply inadequate.

Thank you all. Every one of you contributed in some way to the refinement of these ideas and this text. This may have been my baby, but it had many midwives.

—Jason McC. Smith  
Seattle  
September 4, 2011

# About the Author

**Jason McC. Smith** received his Ph.D. in computer science in 2005 from the University of North Carolina at Chapel Hill, where the Elemental Design Patterns were born as part of the System for Pattern Query and Recognition project. Dr. Smith has been awarded two U.S. patents for research performed at UNC-CH, one for technologies related to SPQR and one for the FaceTop distributed document collaboration system.

Prior to that, Dr. Smith spent many years in industry as a physics simulation engineer and consultant building off of dual B.Sc. degrees in physics and mathematics from the University of Washington. Projects of note included sonar and oceanic environment simulation, electronic engineering simulation, commercial and military aircraft flight simulation, and real-time graphical training systems.

Four years at IBM Watson Research provided Dr. Smith with an opportunity to apply the lessons of SPQR and the EDP catalog and compositional approach to immense bodies of software, both legacy and modern.

Dr. Smith is currently Senior Research Scientist at The Software Revolution, Inc., in Kirkland, Washington, where he continues to refine the EDP catalog and look for ways to enhance the company's goal of automated modernization and transformation of legacy systems.

*This page intentionally left blank*

---

# Introduction to Design Patterns

---

Design patterns are one of the most successful advances in software engineering, by any measure. The history of design patterns is a strange one though, and somewhere along the way, much of their original utility and elegance has been forgotten, misplaced, or simply miscommunicated. This book can fill in some possible gaps for those who have experience with design patterns and can provide students new to the literature a better way of consuming it bite by bite. When it comes down to it, the design patterns literature as it stands is a collection of rather large nuggets of information of varying degrees of digestibility. This text is a foundation that provides a practitioner familiar with design patterns a methodology for placing those nuggets into a larger system of understanding and provides the student new to design patterns an approach for learning them from basic principles and in smaller pieces that make sense individually. The Elemental Design Patterns are truly elemental in that they form a foundation for design patterns as a discipline.

The collective wisdom of the software engineering community is one of our most valuable assets, and we still have much to learn from each other. This book and the research on which it is based are an attempt to bring to light some of what we have lost regarding design patterns. In the process, it helps fulfill the original intent

of design patterns by establishing a better mechanism for shared discussions of patterns, giving us a richer understanding of the software we produce and consume. Our community has produced a breadth of design patterns, but what we lack is depth. That is, we have a broad understanding of wide areas but only a weak ability to stitch them together into a comprehensive whole. It reminds me of the historical transition from alchemy to modern chemistry—until the periodic table came along, the collective wisdom of many intelligent researchers was precise but not strongly correlated. Arguably, the biggest impact of Dmitri Mendeleev’s original periodic table was not so much that it provided a way for chemists to identify patterns between the building blocks of matter but that it provided a way to use those patterns to predict properties of then-undiscovered elements. Gallium and germanium were the first examples of this, with Mendeleev accurately describing their chemical and physical properties well before their discovery. The periodic table advanced chemistry from descriptive discipline to predictive science.

The emergence of design patterns within the software engineering community began with the publication of the seminal *Design Patterns: Elements of Reusable Object-Oriented Software* in 1995. The Gang of Four (or GoF), Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, gathered the various collected wisdoms that had been percolating in the research and academic communities following Gamma’s 1991 PhD dissertation. That work drew heavily from Christopher Alexander’s earlier work in the 1960s. Alexander was a civil engineer and architect, and his work focused on finding patterns of solutions to particular sets of forces within particular contexts. His primary insight was that there are two types of design that occur within architecture, what he named *unselfconscious* and *selfconscious*.

Unselfconscious design is most often seen in so-called primitive cultures: a house design is copied faithfully, every time, and apprenticeships are used to ensure fidelity and faithfulness to that particular design. This design changes rarely, and adherence to a given form is considered the goal, primarily because the particular design is the distillation of centuries or millennia of trial and error—it works, and if it ain’t broke, don’t fix it. While the problem of providing people with housing is universal, the various contexts in which that problem occurs, such as rain, desert, ice, swamp, and forest, give rise to an amazing array of styles and designs, but within a particular context, a single design may be considered “the only solution,” and it is frequently incredibly effective given its specific environmental needs. The design, however, is applied without much, if any, individual discretion or decision making.

Selfconscious design is a more modern invention; the designer is free to make conscious decisions at almost every turn involving style, aesthetics, and materials. This architectural freedom can be seen in the wide array of modern architecture

within a given locale, city, or neighborhood. Even on your own street, you are likely to see a plethora of styles and distinctive flourishes, each the result of many conscious decisions on the part of the architect. The modern designer has a wide palette of styles to choose from, and generally speaking, the only problem is balancing the owner's aesthetics and wallet. Sure, the houses meet the basic criteria of putting a roof over the occupants' heads, but that's a pretty low bar to set when there are so many other axes on which a design can be examined. When a designer is freed to do anything, it becomes even harder to pick the effective solutions out of the nearly infinite set of inappropriate or just plain bad ones. Building codes are one way we try to limit the bad choices in housing design, but even given those as a starting point, the task is daunting. Merely reading building codes and adhering to them is not going to produce an effective work of architecture. Building codes are generic, but good architecture takes into consideration the environment at every level of detail, from global latitude and regional weather patterns to local soil grades and site-specific terrain or foliage.

You can see the results of this selfconscious design in almost any town or city. One house may be Georgian, one pseudo-Victorian, another a modern glass and steel box, or perhaps a split-level, a ranch, or any other number of styles, kinds, and types of construction, materials, and architectures. We have to ask ourselves, however, whether these designs work as optimal, or even just effective, solutions for that particular environment, for that particular context. Austin, Texas, for example, may not be the best place to build an unshaded glass-faced edifice because the sun is so intense in the summer, creating an added expense from the large increase in cooling costs. Upstate New York may not be an appropriate place for a flat roof because the weight of many feet of snow in the dead of winter adds a significant load to the room beams. The environmental context, the set of forces that create the situation in which the general problem of designing appropriate housing must be solved, is frequently ignored, and the solutions are generally only minimally satisfying or give rise to new problems that must be addressed.

It should be apparent how this applies to software engineering: we are capable of doing nearly anything that pops into our heads, even more so than the architects of physical buildings. This is the amazing strength of programming—and its Achilles heel. We can do just about anything, and usually manage to do so, but unfortunately, the subset of good things out of the set of anything is quite small, and our projects are often late, over budget, and frequently fail in ways spectacular and quiet. Rarely do we walk away from projects with a feeling of accomplishment—more often, we feel we dodged a bullet. Again. Why is this? Why, when we have decades of collective experience, and quite possibly millions of tallied person-years

in the field, are we still thrashing in the weeds every time we approach a problem? Some designers and developers seem to be phenomenally able to sidestep the complexity and find the kernel of effectiveness in a design. The rest of us seem to be perpetually stuck between the unselfconsciousness of “because I was told to” and the paralysis of selfconscious design.

Alexander’s work was an attempt to alleviate this problem for architects of buildings, to bring to light the disparity between the effectiveness of the primitive cultures at design and the nearly spastic try-anything approach of modern architecture. Somewhere in between is a balance to be struck. We need to find the underlying principles and general solutions that exist in unselfconscious architecture and describe them in a way that makes them applicable in a wide variety of contexts selfconsciously and with deliberate intent. The wisdom of the various attempts at solutions, hard-earned through trial and error, need to be distilled into a body of concepts that can be learned by anyone, applied in numerous places, and used as a guide for thinking about design.

This is what design patterns are—the distillation of expertise by an exuberant and robust community. This is crowdsourcing at its best. The patterns community that has grown over the decade-plus since the original GoF work is large and energetic, and our output is voluminous. Grady Booch and Celso Gonzalez have been collecting every pattern they can find in industry and academia at their website [11]. So far, they have over 2,000 of them. The quantity of output in this community is huge, and although there are some discussions about the quality, the more pressing problem is one of scale.

Even with a fully indexed, well-curated collection of quality design patterns, there is simply too much information for a nonscholar to sift through accurately and quickly. Worse, it is incredibly difficult for a student wishing to learn the principles behind good design to do so solely from examples of good design. It is a bit like trying to learn the mathematics of aeronautical flow from inspecting aircraft on a runway. For experienced patterns practitioners who believe they have uncovered a new design pattern, there’s no ready way to compare a new pattern against existing patterns to see how it relates to the established literature, and there’s no way to create tooling to support this need.

What the software development community needs is a more thorough understanding of what it has at its disposal, a methodology that explains how to more precisely describe the existing design patterns and does so using components and well-defined principles that are accessible to the student or new developer. What we need is a guide to the underlying basic principles of our design patterns literature

so that we can better comprehend, teach, and learn our identified best practices. This book is a foundation for that guide.

## 1.1 Tribal Musings

The efficiencies we gain from documenting and passing along known best practices are important, but the reason we must do so has been largely ignored in our community. To put it bluntly, we are mortal, and our young field is aging. Already we have lost a number of luminaries who established the groundwork for our industry, and many more will be gone soon. It is just a fact of life, one that we are poorly prepared to deal with as a discipline.

Worse still, software has a peculiar trait of living long past its expected lifetime. COBOL is still a force to be reckoned with in business systems around the globe. Fortran still performs much of the computation in the world's scientific modeling software. Currently shipping major high-performance computer systems have code embedded deep in their firmware that was first created three decades or more ago, in assembler or C. You can be almost certain that somewhere in the millions of lines of implementation that came with your latest personal computer acquisition lies a piece of source code that no person currently understands.

We know we should document our software; we know we should keep it up to date; we know we should commit to pen or screen the whys, the hows, and the reasons; but we also know it is a pain. It really is, so we don't do it. What we have instead is a body of knowledge that is locked within the heads of developers, that is passed along in fits and spurts, when prompted and only where necessary, frequently without any comprehensive framework of common understanding among stakeholders.

Grady Booch has popularized the phrase "tribal knowledge" for such information [10], and it fits all too well. It also has some rather unsettling corollaries.

Cultures that rely solely on oral tradition for the passing of knowledge are limited in both bandwidth and accuracy, and that's assuming they have a *strong* tradition of passing along the information. Cultures with a weak discipline for veracity and precision in information transfer leave themselves open to more rapid corruption. A strong oral tradition, however, can result in a very different outcome.

The development community has what is ultimately an oral tradition of information transfer. Although we may write down bits and pieces of what we understand, we frequently do not write down the entirety of our comprehension, and we do not keep such documentation in sync with the evolution of our systems. This



document rot is pervasive, and only by asking around for further information can we hope to fill in the gaps to find out why a particular system is how it is.

This isn't always seen as a bad thing, to be honest. Agile software development methodologies prefer working code over documented code, and it's hard to argue with this viewpoint. Until it matters, of course. Agile systems have a funny way of becoming legacy systems, of growing into mature codebases with larger teams that must work in concert. Eventually, code that started as an agile effort, if it is successful, will face many of the same challenges as traditionally developed systems. Developers leave. Documentation rots. Knowledge is lost.

Software as it currently stands is not what anyone could accurately call self-documenting, and extracting the salient reasons why a thing was done in a particular way, directly from the source code, has been considered nearly impossible for an automated system. This is unfortunate, because we would like to have our cake and eat it too. We want up-to-date documentation when and where we need it, but we don't want to be burdened with it otherwise. We'd like our code to be much more self-documenting, or at least automatically documentable, but most of us don't have that luxury. So we punt and hope for the best. Meanwhile, our collective understanding of the system degrades. In the end, what we have is best described as a very weak oral tradition.

The result is that the collected tribal knowledge degrades into "tribal mythology." "Why?" is not a question that can be answered any longer, except to say, "Because we've always done it that way." I have a sneaking suspicion that if you have ever been the new hire on a development team, you're nodding in horror right now. You've had that discussion in real life, probably more times than you care to recall.

Tribal mythology is action without comprehension. It is rote without any foundation on which to state why. Other indications that tribal mythology is active in a group include the following: "Because that's how I was taught it." "I'm not really sure, but Joe says that's how its done." "Jane could have told you, but she retired last year, so just copy what's there." "Oh no, don't change that! It'll break and we won't be able to fix it." These comments exhibit a failure to comprehend the reasons behind an action, or at least an unwillingness or inability to pass the comprehension along to the listener. Over time, this lack of understanding breeds a great deal of uncertainty and fear of change. Unfortunately, it is at some level the status quo on most projects, which is ironic given that our industry is driven by innovation, change, and advancement of the state of the art.

Tribal wisdom, however, is the virtuous flip side of this tribal mythology. It is prescribed action *with* understanding, *how* accompanied by *why*, and is adaptable

to new environments, new situations, and new problems. It transcends rote copying, and provides illumination through a comprehensive discussion of the reasons behind the action. At some point in the past, for almost every action or decision in a system, someone knew why it was done that way. Carrying those decisions forward, even the small ones, can be critical. Small decisions accrete into large systems, and small designs build into large designs. By ensuring that we have a strong tradition of knowledge retention that facilitates understanding, we build a tradition of tribal wisdom.

Tribal wisdom is what design patterns were intended to collect. Sadly, they are frequently (mis)treated as tribal mythology, by applying the *how* without a clear comprehension of the *why*.

If you haven't yet had the pleasure of running into this situation in your career, let me offer another example that may be illuminating. Recently my wife and I bought our first house, and with it, our first yard. The region we live in is renowned for its rain and consequently its moss. Now, I like moss. It's green, it takes about zero maintenance, and it makes a nice soft ground cover. It satisfies all the usual requirements for a yard, with less work than grass requires, but we had an odd situation. Part of the yard is heavily shaded and rarely, if ever, sees sun. This area is basically solid moss, with no grass or any other vegetation. Even shade-tolerant grasses can't get enough light to thrive.

Twenty feet on either side of the heavily shaded portion, however, sunlight is available on most days when the sun is actually out. Moss grows in patches through this section, but in my initial assessment, I thought it was fine. The moss and grass were coexisting nicely, and the moss wasn't choking out the grass, merely filling in the places where the grass wasn't quite so thick. In the sunniest areas, there was almost no moss but lots of grass. In the shadiest areas, there was solid moss but no grass. In the transitional regions, the two coexisted. What could be better?

Unfortunately, long-time residents who saw this situation were horrified. "You have to get rid of all the moss!" When I asked why, I was met with answers such as "Because it's not grass." "Because it's what you *do*." "Because it's bad for the lawn." No one could tell me, to my satisfaction, *why* I should get rid of the moss. It seemed to me that if I removed *all* of the moss, in all areas, regardless of the local micro-environment, I would have a bare spot where grass wouldn't grow in the shade. This was less than optimal.

To make matters worse, as is the case in many software projects, I had inherited a situation in which I had no idea what the previous residents had done for maintenance in the yard or why. There was no documentation to indicate what I should do for my lawn or why the yard had been left in this configuration. So I ran a couple of

experiments. In the shadiest areas, I pulled up a small section of moss and seeded it with grass. In the rest of the yard, I let the moss go to see what would happen.

The grass seed in the shadiest area never thrived. Some sprouted, but it could never get established well. Applying the tribal mythology would have resulted in bare dirt in a good section of my yard, and frankly, I prefer the moss to that. It is green and lush, and it thrives in that area without maintenance. For that specific area, moss is a good ground cover solution.

In the rest of the yard with little or no shade, it turns out that moss and grass *do* play well together, more or less. The grass grows nicely, and the moss can't overcome it directly. Unfortunately, the moss has a side effect. In the sunniest areas, the moss acts as a protective layer for weeds to sprout underneath, safe from birds and mice who would eat the seeds or shoots, and properly moist. The moss won't choke out the grass, but the weeds definitely will. By letting the moss exist in the sunny areas, I was giving weeds a nursery to get established, and when they penetrated the moss, they thrived in the sunlight and spread rapidly. The moss also provided a protective moist layer for the roots of weeds to travel along, offering them an unhindered growth channel.

In the shadiest areas, this wasn't a problem, because there simply isn't enough sun for grass *or* weeds to grow well, but in the sunny areas, it was horrible. Within a couple of months, I was fighting a literal turf war with the weeds. The moss was never a problem by itself, but it set the scene for a much larger one.

And now I know the *why* behind removing moss from a lawn. It's not so much the moss that is the issue, it is that the moss creates a secondary microclimate that sets up a serious situation. Essentially, the moss creates a new set of forces at play that form a new *context*. Within that new context, that new environment, new problems arise—like weeds. Now the advice to remove the moss makes sense, at least for areas where weeds are an issue, such as the sunny areas of my yard.

Because I know the why, I can now alter my application of this knowledge according to the environmental forces. In the sunny areas, I must remove and prevent the moss so that weeds are not a later problem. In the shady areas, I choose not to because doing so would create another problem for me, leaving me with bare dirt where I'd struggle to get grass to grow well.

As it is, because I know the reasons behind the advice, I can custom fit the solution I was given—"Remove the moss"—based on the context—sunny vs. shady—and not only solve my initial problem but prevent new ones from being created. What was initially tribal mythology is now tribal wisdom that can be shared, adjusted, and applied when and where appropriate. In essence, it is the beginning of a pattern.

## 1.2 Art or Science?

There is no doubt that patterns are a thriving meme, and one with great utility. Entire academic conferences are now dedicated to patterns, Ackerman and Gonzalez’s patterns-based engineering is becoming a defined discipline in its own right [2], and industry consultants are now expected to have them under their belt and be able to whip out Unified Modeling Language (UML) diagrams of them on the spot. Tools exist to produce, display, generate, and extract patterns. Patterns, as a collective whole, are an assumed component of the software engineering landscape. We’re just not quite sure how they fit into that landscape or how they fit with each other. Two issues prevent a more comprehensive approach to patterns, and unfortunately they are ubiquitous in the industry. The first is treating patterns as frozen elements to be blindly copied, the second is confusing language-specific pattern implementations with variants of the patterns themselves.

### 1.2.1 Viewing Patterns as Rote

Ask a dozen developers to define design patterns, and you’ll likely get a dozen answers. Among the more traditional “a solution to a recurring problem within a particular context” answers, you’re also likely to hear phrases such as “a recipe” or “an example structure” or “some sample code,” betraying a rather narrow view of what patterns provide. Patterns are intended to be mutated, to be warped and molded, to meet the needs of the particular forces at play in the context of the problem, but all too often, a developer simply copies and pastes the sample code from a patterns text or site and declares the result a successful application of the pattern. This is usually a recipe for failure instead of a recipe for producing a good design.

Pure rote copying of the structure of the pattern “because this authority says so” is a reversion to Alexander’s concept of unselfconscious design. We undermine the entire purpose of design patterns when we do that. We need to be able to describe the whys behind a pattern as well as the hows. Without the understanding of the reasons that led to the description of that pattern, rote application often results in misapplication. At best, the result is a broken pattern that simply does not match the intended outcome. At worst, it injects an *iatrogenic* pattern into the system—one that is intended and thought to be of benefit but instead produces a malignant result that may take years to uncover. It doesn’t just fail to provide the expected enhancement, it actively creates a new problem that may be worse than the original one. This is patterns as tribal mythology—action without understanding.

The traditional design pattern form, as defined in *Design Patterns* [21], explains the whys behind a pattern—motivations, applicability, and consequences—but it is up to the reader to tease out the underlying concepts that form a pattern. To some degree, these subconcepts are described in the Participants (what are the pieces) and Collaborations (how do they relate) sections for each patterns, but again, these are frequently treated by developers as checklists of pieces of the solution for rote implementation instead of as a description of the underlying concepts and abstractions that comprise a solution.

## 1.2.2 Language-Dependent Views

Ask a developer how important patterns are to his or her work, and frequently the answer will be based on the implementation language the developer is using. This isn't surprising. Different languages offer different strengths centered around the concepts they support and how they express them. How those concepts happen to be expressed is more often the start of flame wars between language fans, but ignoring the underlying concepts leads to much argument over nothing of particular consequence in most cases. Whether blocks are delineated by curly braces, as in the C family, or by whitespace, as with Python, isn't nearly as important as having the concept of blocks in the first place.

What this means is simply that some patterns are easier to implement in some languages than in others. In fact, some languages can make the concepts behind certain patterns so simple to implement that they're known as language features. The *Visitor* pattern is a good example.<sup>1</sup> *Visitor's* Implementation section [21, pg. 338] says, "*Visitor* achieves [its goal] by using a technique called double-dispatch. It's a well-known technique. In fact, some programming languages support it directly (CLOS, for example)." What does this mean? It means that mentioning the *Visitor* design pattern to CLOS (Common LISP Object System) developers will leave them scratching their heads. "A pattern? For a language feature? Why?" In CLOS, *Visitor* is essentially built in. You don't need a pattern to tell you how to best express the concept—it's already there in the language as a basic feature. In most other languages, however, *Visitor* provides a clean way of expressing the same programming concept of double dispatch.

This illustrates an important point. If you mention double dispatch instead of the *Visitor* pattern to the same CLOS developers, they would know what you mean, how to use double dispatch, and when not to use it. Terminology, particularly shared common terminology, matters a great deal.

---

1. You don't need to know what the *Visitor* pattern is right now. I selected it only because the discussion of *Visitor* explicitly addresses the point I'm making.

This is true for all languages and all patterns: some languages make certain patterns easier or trivial to implement and other patterns more difficult to realize. No language can really be considered superior to another in this case, however. One common myth is that design patterns make up for flaws in programming languages, but that isn't the case. Design patterns describe useful concepts, regardless of the language used to implement them. Whether a specific concept is baked into the feature set of a language or must be implemented from scratch is irrelevant. The concept is still being expressed in the implementation, and that is the critical observation that lets us talk about software design independently of software implementation. Design is concepts; how those concepts are made concrete in a given language is implementation.

When you get down to it, there's no reason you couldn't implement every pattern in the GoF text in plain C—but it would be extremely tedious. You'd have to build up best practices for binding data and functions into meaningful semantic units, encapsulating that data, ensuring that data is ready to use at first accessibility, and so on. This sounds like a lot of work, but these were concepts considered *so* important that they launched a revolution in language features to make them easier to work with. That revolution was object-oriented programming.

In object-oriented languages, those concepts are included as primary language features called classes, visibility, and constructors. Again, we can refer to the GoF: “If we assumed procedural languages, we might have included design patterns called ‘Inheritance,’ ‘Encapsulation,’ and ‘Polymorphism.’” The authors felt that this statement was important enough that it appears in Section 1.1 in the Introduction. And yet again, this is a fundamental point that seems lost on most developers, so let me restate it.

Patterns are language-independent concepts; they take form and become concrete solutions only when you implement them within a particular language with a given set of language features and constructs.

This means that it is a bit strange to talk about “Java design patterns,” “C++ design patterns,” “WebSphere design patterns,” and so on, even though we all do it. It's a mildly lazy form of shorthand for what we really mean, or should mean: design patterns as implemented in Java, C++, WebSphere, and so on, regardless of language or API.<sup>2</sup>

Unfortunately, if you're like many developers who have encountered one of the multitude of books on design patterns, you may have been trained, or at least have been erroneously led to believe, that there is some ephemeral yet fundamental

---

2. Some design patterns are unique to specific languages, and only those languages, but those patterns are often called *language idioms*. In this text when we use the term *design patterns*, we are specifically talking about concepts that are language independent.

difference between patterns as expressed in Java and those expressed in another language such as Smalltalk. There really isn't. The concepts are the same; only the manner in which they are expressed and the ease with which a programmer can implement them in that specific language differ.

We need to focus on these when investigating design patterns, and these abstractions must be the crux of understanding patterns. Unless we make the effort to look at patterns as language-independent concepts, we are merely learning rote recipes again and losing much of what makes them so useful.

### 1.2.3 From Myth to Science

The issues described previously belie an underlying problem with design patterns as they are often conveyed, used, and understood today. All too often, we still don't know why we do what we do, even when we use design patterns in our code. By using design patterns so inflexibly, we've simply better documented a body of unselfconscious snippets without the comprehension that comes from a methodical analysis of the snippets.

We have an art. What we need is a science. After all, we throw around the terms *computer science* and *software engineering* with abandon. Treating patterns as sample code misses the point of design patterns. Design patterns enable us as an industry to experiment with those concepts and share, discuss, and refine our findings.

Patterns as rote recipes are tribal mythology.

Patterns as concepts are the foundations of a science.

Elemental Design Patterns are the building blocks of that science.

# Index

## A

Abadi, Martín, 281  
*Abstract Factory* pattern  
  as coalesced PINbox, 65  
  as collapsed PINbox, 50  
  as expanded PINbox, 64  
  as stacked PINbox, 60  
  as UML, 59  
  description, 260–263  
  rho-calculus, 313  
*Abstract Interface* pattern  
  applicability, 141–142  
  as part of *Fulfill Method* pattern, 68–72  
  collaborations, 142  
  consequences, 142–143  
  creating objects, 43–44  
  implementation, 143–144  
  intent, 140  
  isotopic forms, 75  
  motivation, 140–141  
  participants, 142  
  related patterns, 144  
  rho-calculus, 298  
  structure, 142  
Abstraction density, EDPs, 110–112  
Ackerman, Lee, 9  
Alexander, Christopher, 2, 4, 86  
Animal example  
  eating. *See* Objectifier pattern.  
  moving. *See* Abstract Interface pattern.  
Anti-patterns, 91  
Array sorting example. *See* Recursion pattern.  
Asynchronous execution, 146

## B

Beck, Kent, 32–33  
Behavioral patterns, 273–279

Booch, Grady  
  pattern collections, 4  
  tribal knowledge, 5–8

Books and publications  
  *Clean Code*, 32  
  *Design Patterns: Elements of...*, 2  
  *Refactoring*, 92  
  *Refactoring to Patterns*, 92

## C

C code examples, 118, 120, 121, 193, 246  
C# code examples, 178  
C++ code examples, 26, 29, 31, 36, 125,  
  128, 138, 143, 148, 153, 160, 166–168,  
  173, 176, 184, 190, 193–195, 201, 210,  
  213, 216, 220, 222, 227, 233,  
  238, 247  
Calls. *See* Delegation pattern.  
Cardelli, Luca, 281  
Categories of pattern definition, 22  
*CEO example*, 36. *See also* Delegation  
  pattern.  
*Chain of Responsibility* pattern  
  description, 273–275  
  rho-calculus, 318  
Chemical isotopes, 72–74  
Cheshire cat technique, 119  
Clarity of expression, EDPs, 109–110  
Classes  
  decomposing patterns, 25–26  
  objectifying similar behaviors, 244. *See also*  
    Objectifier pattern.  
*Clean Code*, 32  
Coalescing PINboxes, 62–65  
Cohesion  
  EDPs, 43–44  
  objects, 35



Collapsed PINboxes, 49–51  
 Common Lisp Object System (CLOS), 10  
 Composing EDPs, 68–77  
 Composition, rho-calculus, 291–293  
*Conglomeration pattern. See also Delegated Conglomeration pattern.*  
   applicability, 161  
   as part of *Factory Method* pattern, 263–265  
   as part of *Template* pattern, 275–279  
   collaborations, 162  
   consequences, 162  
   design space, 35–38  
   implementation, 162  
   intent, 159  
   method call classification, 163–164  
   motivation, 159–161  
   participants, 161–162  
   related patterns, 162–163  
   rho-calculus, 300  
   structure, 161  
 Context, EDPs, 30–33  
 Context category, 22  
*Create Object* pattern  
   applicability, 122  
   as part of *Abstract Factory* pattern, 261–263  
   as part of *Factory Method* pattern, 263–265  
   collaborations, 123  
   consequences, 124  
   implementation, 124–125  
   intent, 117  
   motivation, 117–122  
   participants, 123  
   related patterns, 125  
   rho-calculus, 295–296  
   structure, 123  
 Creational patterns, 260–265

## D

Data protocol fallback example. *See Revert Method* pattern.  
 Decomposing Message. *See Conglomeration* pattern.  
 Decomposing patterns  
   categories of pattern definition, 22  
   classes, 25–26  
   context category, 22  
   example, *Decorator* pattern, 18–21  
   fields, 25–30

  goal of, 22  
   methods, 25–30  
   minimum size, 21–30  
   object-oriented programming, entity interactions, 28  
   objects, 25–30  
   problem category, 22  
   scoping, 22–24  
   solution category, 22  
   types, 25–30  
   unnamed namespaces, 24  
*Decorator* pattern  
   as collaboration UML element, 47  
   as expanded PINbox, 57  
   as PINbox, 52  
   description, 265–269  
   rho-calculus, 316  
*Decorator* pattern, examples  
   decomposing patterns, 18–21  
   recreating, 77–90  
*Defer Implementation* pattern. *See Abstract Interface* pattern.  
*Delegated Conglomeration* pattern. *See also Conglomeration* pattern; *Delegation* pattern; *Deputized Delegation* pattern; *Trusted Delegation* pattern.  
   applicability, 189  
   collaborations, 190  
   consequences, 190  
   implementation, 190–191  
   intent, 187  
   method call classification, 191–192  
   motivation, 187–189  
   participants, 189–190  
   related patterns, 191  
   rho-calculus, 303  
   structure, 189  
*Delegation* pattern. *See also Delegated Conglomeration* pattern; *Deputized Delegation* pattern; *Trusted Delegation* pattern.  
   applicability, 147  
   collaborations, 148  
   consequences, 148  
   design space, 35–38  
   implementation, 148  
   intent, 145  
   method call classification, 149–150

- motivation, 145–146
  - participants, 147
  - related patterns, 149
  - rho-calculus, 299
  - structure, 147
  - Delegation patterns
    - Delegated Conglomeration*, 187–192
    - Delegation*, 145–150
    - Deputized Delegation*, 216–221
    - Trusted Delegation*, 200–208
  - Deputized Delegation* pattern. *See also*
    - Delegated Conglomeration* pattern;
    - Delegation* pattern; *Trusted Delegation* pattern.
  - applicability, 218
  - collaborations, 219
  - consequences, 219–220
  - implementation, 220
  - intent, 216
  - method call classification, 220–221
  - motivation, 216–218
  - participants, 218–219
  - related patterns, 220
  - rho-calculus, 306
  - structure, 219
  - Deputized Redirection* pattern. *See also*
    - Redirected Recursion* pattern;
    - Redirection* pattern; *Trusted Redirection* pattern.
  - applicability, 224
  - as part of *Proxy* pattern, 269–273
  - collaborations, 226
  - consequences, 227
  - implementation, 227
  - intent, 222
  - method call classification, 227–228
  - motivation, 222–224
  - participants, 224–226
  - related patterns, 227
  - rho-calculus, 307
  - structure, 226
  - Design patterns. *See also* EDPs (Elemental Design Patterns).
    - anti-patterns, 91
    - collections of, 4
    - definition, 21
    - diagramming. *See* PIN (Pattern Instance Notation); PINboxes.
    - iatrogenic, 91
    - instance notation and roles, 293–295
    - malignant, 91
    - partial, 91
    - recognizing, 15–17
    - as rote, 9–10
  - Design patterns, history of
    - documentation, 5–8
    - founders, 2, 4
    - language-dependent views, 10–12
    - from myth to science, 12
    - pattern collections, 4
    - patterns as rote, 9–10
    - selfconscious design, 2–3
    - seminal works, 2
    - tribal knowledge, 5–8
    - types of design, 2–3
    - unselfconscious design, 2–3
  - Design Patterns: Elements of...*, 2
  - Design space, EDPs, 33–42
  - Diagramming design patterns. *See* PIN (Pattern Instance Notation); PINboxes.
  - Documentation
    - history of design patterns, 5–8
    - and training, 108–109
  - Dot operators, 283
  - Double-dispatch, 10
  - d-pointer technique, 119
- ## E
- EDPs (Elemental Design Patterns). *See also*
    - Design patterns.
    - abstraction density, 110–112
    - anti-patterns, 91
    - background, 14–17
    - breaking into smaller parts. *See*
      - Decomposing patterns.
    - clarity of expression, 109–110
    - cohesion, 43–44
    - combining with other patterns, 101–105
    - composing patterns, 68–77
    - concepts vs. constructs, 16
    - context, 30–33
    - definition, 18
    - design space, 33–42
    - documentation, 108–109
    - field usage, 42–44
    - formalisms, 287–291

EDPs (*Cont.*)

- iatrogenic patterns, 91
  - isotopes, 72–77
  - malignant patterns, 91
  - method call reliance, 29–33
  - method similarity, 32
  - metrics, 109–112
  - partial design patterns, 91
  - pattern specifications, 16–17
  - procedural analysis, 112
  - recognizing patterns, 15–17
  - recreating *Decorator*, example, 77–90
  - refactoring, 91–100
  - reliances, 29–33, 42–44
  - rho-calculus. *See* Rho-calculus, EDP definitions.
  - SPQR (System for Pattern Query and Recognition), 14–17
  - state changes, 42–44
  - training, 108–109
  - unique traits, 13–14
  - used-by relationships, 101–102
- EDPs (Elemental Design Patterns), objects
- Abstract Interface* pattern, 43–44
  - cohesion, 35
  - creating, 43–44
  - decomposing patterns, 25–30
  - equivalence, 34–35
  - Inheritance* pattern, 43
  - instantiating, 43–44
- Encapsulation
- similarity, 32
  - of data. *See* *Create Object* pattern.
  - of design, 101
- Equivalence, objects, 34–35
- Executive* pattern. *See* *Delegation* pattern.
- Expanded PINboxes, 55–56
- Extend Method* pattern
- applicability, 183
  - as part of *Decorator* pattern, 83–86, 268–269
  - as part of *Chain of Responsibility* pattern, 273–275
  - collaborations, 184
  - consequences, 184
  - implementation, 184–185
  - intent, 181
  - method call classification, 185–186

- motivation, 181–182
- participants, 183
- related patterns, 185
- rho-calculus, 302
- structure, 183

*Extending Super* pattern. *See* *Extend Method* pattern.

**F**

- Factory Method* pattern
- description, 263–265
- Fencepost error, 132–134
- Field usage, EDPs, 42–44
- Fields, decomposing patterns, 25–30
- Flyweight* pattern
- as PINbox, 53
- Formalism of patterns, 105–108. *See also* Rho-calculus.
- Founders, history of design patterns, 2, 4. *See also specific names.*
- Fowler, Martin, 92
- Fulfill Method* pattern
- applicability, 231
  - as part of *Abstract Factory* pattern, 261–263
  - as part of *Factory Method* pattern, 263–265
  - as part of *Objectifier* pattern, 77, 79–80, 248–249
  - as part of *Proxy* pattern, 269–273
  - as part of *Template* pattern, 275–279
  - collaborations, 232
  - consequences, 233
  - implementation, 233–234
  - intent, 231
  - motivation, 231
  - participants, 231
  - related patterns, 234
  - rho-calculus, 308
  - structure, 232

**G**

- Gamma, Erich, 2
- Gang of Four patterns
- Abstract Factory*, 260–263, 313
  - behavioral, 273–279
  - Chain of Responsibility*, 273–275, 318
  - creational, 260–265
  - Decorator*, 265–269, 316

*Factory Method*, 263–265, 314–315  
*Proxy*, 269–273, 317  
 rho-calculus, 313–319  
 structural, 265–273  
*Template Method*, 275–279, 319  
 Gonzalez, Celso, 4, 9

## H

Helm, Richard, 2  
 Helper Methods. *See Conglomeration* pattern.  
 History of design patterns  
   documentation, 5–8  
   founders, 2, 4. *See also specific names.*  
   language-dependent views, 10–12  
   from myth to science, 12  
   pattern collections, 4  
   patterns as rote, 9–10  
   selfconscious design, 2–3  
   seminal works, 2  
   tribal knowledge, 5–8  
   types of design, 2–3  
   unselfconscious design, 2–3

## I

Iatrogenic patterns, 9–10, 91  
 Inheritance, multiple, 136  
*Inheritance* pattern  
   applicability, 135  
   as part of *Fulfill Method* pattern, 68–72  
   collaborations, 135  
   consequences, 135–138  
   creating objects, 43  
   implementation, 138–139  
   intent, 130  
   isotopic forms, 76  
   motivation, 130–134  
   participants, 135  
   related patterns, 138  
   rho-calculus, 298  
   structure, 135  
 Instances, creating, 235–239. *See also Retrieve* *New* pattern.  
 Instantiating objects, 43–44  
*Instantiation* pattern. *See Create Object* pattern.  
*IsA* pattern. *See Inheritance* pattern.  
 Isotopes, 72–77, 285–286

## J

Java code examples, 27, 36, 38, 126, 127, 129, 133–134, 141, 144, 145, 152, 156, 162, 169, 184, 187, 188, 197, 206, 233, 243  
 Johnson, Ralph, 2

## K

Kerievsky, Joshua, 92

## L

Language idioms, 11  
 Language-dependent patterns, 10–12

## M

Malignant patterns, 91  
 Martin, Robert, 32  
 Mathematics of patterns, 105–108. *See also* Rho-calculus.  
*Memory management*. *See Object* *management*.  
*Messaging* pattern. *See Delegation* pattern.  
 Method call reliance, 29–33  
*Method Invocation* pattern. *See Delegation* pattern.  
 Method Invocation patterns  
   *Conglomeration*, 159–164  
   *Delegated Conglomeration*, 187–192  
   *Delegation*, 145–150  
   *Deputized Delegation*, 216–221  
   *Deputized Redirection*, 222–228  
   *Extend Method*, 181–186  
   *Recursion*, 165–171  
   *Redirected Recursion*, 193–199  
   *Redirection*, 151–158  
   *Revert Method*, 172–180  
   *Trusted Delegation*, 200–208  
   *Trusted Redirection*, 209–215  
 Methods  
   decomposing patterns, 25–30  
   naming, 32–33  
   overloaded, 32  
   previously extracted, implementing, 231–234. *See also Fulfill Method* pattern.  
   similarity, 32  
 Metrics for EDPs, 109–112  
 Moss example, 7  
 Meyers, Scott, 124  
 Multiplicity connections, PINboxes, 56–62

## N

Namespaces, unnamed, 24  
 Naming conventions, 18

## O

Object Elements patterns  
   *Create Object*, 117–125  
*Object management*  
   C++ `shared_ptr`, 240  
   C++ unmanaged, 235  
   garbage collection, 235, 238, 240, 243  
   *Objective-C autorelease*, 240  
*Object Recursion* pattern. *See also Recursion*  
   pattern; *Redirected Recursion* pattern.  
   applicability, 252–253  
   as part of *Decorator* pattern, 83–86,  
     268–269  
   collaborations, 255–256  
   consequences, 256  
   implementation, 256–257  
   in *Inheritance*, 133  
   intent, 251  
   motivation, 251–252  
   participants, 253  
   related patterns, 257  
   rho-calculus, 312  
   structure, 253, 254–255  
*Objectifier* pattern  
   applicability, 245–247  
   collaborations, 249  
   consequences, 249–250  
   implementation, 250  
   intent, 244  
   motivation, 244–245  
   participants, 249  
   related patterns, 250  
   rho-calculus, 311  
   structure, 248–249  
 Objective-C code examples, 37, 131, 132, 137,  
   156, 190  
 Object-oriented programming, entity  
   interactions, 28  
 Objects  
   *Abstract Interface* pattern, 43–44  
   cohesion, 35  
   *Create Object* pattern, 117–125  
   creating, 43–44  
   decomposing patterns, 25–30

  equivalence, 34–35  
   hiding details. *See Proxy* pattern.  
   *Inheritance* pattern, 43  
   instantiating, 43–44  
   limiting access. *See Proxy* pattern.  
   *Retrieve* pattern, 126–129  
   shared, referencing without owning,  
     240–243. *See also Retrieve Shared*  
     pattern.  
 Off-by-one error, 132–134  
 Opaque pointer technique, 119  
 Overriding a method  
   in *Inheritance*, 133  
   requiring. *See Abstract Interface* pattern.  
   without replacing. *See Extend Method*  
     pattern.

## P

Painter example. *See Redirection* pattern,  
   *Conglomeration* pattern.  
 Paratroopers example. *See Redirected*  
   *Recursion* pattern.  
 Partial patterns, 91  
 Pattern specifications, 16–17  
 Patterns. *See Design* patterns.  
 Peeling PINboxes, 62–65  
 PIN (Pattern Instance Notation), 45–49  
 PINboxes  
   coalescing, 62–65  
   collapsed, 49–51  
   definition, 49  
   expanded, 55–56  
   multiplicity, 56–62  
   peeling, 62–65  
   stacked, 56–62  
   standard, 51–55  
 Pointer-to-implementation (pimpl) technique,  
   119  
 Polymorphism. *See Abstract Interface* pattern;  
   *Object Recursion* pattern.  
*Polymorphism* pattern. *See Abstract Interface*  
   pattern.  
 Problem category, 22  
 Procedural analysis, EDPs, 112  
*Proxy* pattern  
   description, 269–273  
   rho-calculus, 317  
 Pure virtual method, 143–144

Python code examples, 129, 138, 144, 162, 182, 197, 206, 233, 238

## R

Recursion, definition, 165

Recursion, patterns

*Object Recursion*, 251–257

*Recursion*, 165–171

*Redirected Recursion*, 193–199

*Recursion* pattern. *See also Object Recursion* pattern; *Redirected Recursion* pattern.

applicability, 168

collaborations, 169

consequences, 169

design space, 35–38

implementation, 169

intent, 165

method call classification, 170–171

motivation, 165–168

participants, 168–169

related patterns, 169–170

rho-calculus, 301

structure, 169

*Redirected Recursion* pattern. *See also*

*Deputized Redirection* pattern; *Object*

*Recursion* pattern; *Recursion* pattern;

*Redirection* pattern; *Trusted Redirection* pattern.

applicability, 196

as part of *Chain of Responsibility* pattern, 273–275

collaborations, 197

consequences, 197

design space, 40

implementation, 197–198

intent, 193

method call classification, 198–199

motivation, 193–196

participants, 197

related patterns, 198

rho-calculus, 303

structure, 196

*Redirection* pattern. *See also Deputized*

*Redirection* pattern; *Redirected Recursion* pattern; *Trusted Redirection* pattern.

applicability, 153–154

collaborations, 155

consequences, 155

design space, 35–38

implementation, 156

intent, 151

method call classification, 157–158

motivation, 151–153

participants, 154

related patterns, 157

rho-calculus, 300

structure, 154

*Redirection* patterns

*Deputized Redirection*, 222–228

*Redirected Recursion*, 193–199

*Redirection*, 151–158

Reduction rules, 285–286, 291–293

Refactoring EDPs, 91–100

*Refactoring*, 92

*Refactoring to Patterns*, 92

Refactorings

between EDPs, 106

Extract Method, 92–93

Move Method, 92, 95, 97–99

Reliance operators, 282–284

Reliances, 29–33, 42–44

Requests, distributed handling. *See Object Recursion* pattern.

*Retrieve New* pattern

applicability, 236

collaborations, 237–238

consequences, 238

implementation, 238–239

intent, 235

motivation, 235–236

participants, 237

related patterns, 239

rho-calculus, 309

structure, 236–237

*Retrieve* pattern

applicability, 127

as part of *Abstract Factory* pattern, 261–263

as part of *Factory Method* pattern, 263–265

collaborations, 128

consequences, 128

implementation, 128–129

intent, 126

motivation, 126–127

participants, 128

related patterns, 129

*Retrieve* pattern (*Cont.*)  
 rho-calculus, 296–297  
 structure, 127

*Retrieve Shared* pattern  
 applicability, 241  
 collaborations, 243  
 consequences, 243  
 implementation, 243  
 intent, 240  
 motivation, 240–241  
 participants, 242  
 related patterns, 243  
 rho-calculus, 310  
 structure, 241–242

*Revert Method* pattern  
 applicability, 177  
 collaborations, 178  
 consequences, 178  
 implementation, 178–179  
 intent, 172  
 method call classification, 179–180  
 motivation, 172–176  
 participants, 177–178  
 related patterns, 179  
 rho-calculus, 301  
 structure, 177

Rho-calculus  
 composition, 291–293  
 dot operators, 283  
 EDP formalisms, 287–291  
 isotopes, 285–286  
 overview, 281–282  
 pattern instance notation and roles,  
 293–295  
 reduction rules, 285–286, 291–293  
 reliance operators, 282–284  
 scoping, 283  
 similarity, 286–287  
 transitivity, 285–286

Rho-calculus, EDP definitions  
*Abstract Interface*, 298  
*Conglomeration*, 300  
*Create Object*, 295–296  
*Delegated Conglomeration*, 303  
*Delegation*, 299  
*Deputized Delegation*, 306  
*Deputized Redirection*, 307

*Extend Method*, 302  
*Inheritance*, 298  
*Recursion*, 301  
*Redirected Recursion*, 303  
*Redirection*, 300  
*Retrieve*, 296–297  
*Revert Method*, 301  
*Trusted Delegation*, 304  
*Trusted Redirection*, 305

Rho-calculus, Gang of Four definitions  
*Abstract Factory*, 313  
*Chain of Responsibility*, 318  
*Decorator*, 316  
*Factory Method*, 314–315  
*Proxy*, 317  
*Template Method*, 319  
 Rho-calculus, intermediate definitions  
*Fulfill Method*, 308  
*Object Recursion*, 312  
*Objectifier*, 311  
*Retrieve New*, 309  
*Retrieve Shared*, 310

## S

Scoping, 22–24, 283  
 Selfconscious design, 2–3  
 Shop foreman. *See* *Redirection* pattern.  
 Sigma-calculus, 282–285  
 Similarity  
   objects, 32  
*Singleton* pattern  
   as collapsed PINbox, 50  
 Social network invites example. *See* *Delegated Conglomeration* pattern.  
 Solution category, 22  
 SPQR (System for Pattern Query and  
   Recognition), 14–17  
 Stacked PINboxes, 56–62  
 Standard PINboxes, 51–55  
 State changes, 42–44  
*Strategy* pattern  
   as pattern:role tags in UML, 48  
   multiple instances as pattern:role tags in  
     UML, 49  
   multiple instances as PINboxes, 61, 62  
 Structural patterns, 265–273  
 Synchronous method calls, 146

**T**

*Template Method* pattern  
 as part of *Factory Method* pattern, 279  
 description, 275–279  
 in sequence diagram, 51  
 rho-calculus, 319

Tom Sawyer. *See* *Redirection* pattern.

Training, 108–109

Transitivity, rho-calculus, 285–286

Tribal knowledge, 5–8

Tribal mythology, 6–9, 12

*Trusted Delegation* pattern  
 applicability, 204  
 collaborations, 205  
 consequences, 205  
 implementation, 205–206  
 intent, 200  
 method call classification, 207–208  
 motivation, 200–204  
 participants, 205  
 related patterns, 206–207  
 rho-calculus, 304  
 structure, 204

*Trusted Redirection* pattern. *See also* *Deputized Redirection* pattern; *Redirected Recursion* pattern; *Redirection* pattern.  
 applicability, 212  
 as part of *Object Recursion* pattern, 79,  
 81–83, 254–256  
 collaborations, 213  
 consequences, 213  
 implementation, 213

intent, 209  
 method call classification, 214–215  
 motivation, 209–211  
 participants, 212–213  
 related patterns, 213–214  
 rho-calculus, 305  
 structure, 212

**Type Relation patterns**

*Abstract Interface*, 140–144  
*Inheritance*, 130–139

*Type Reuse* pattern. *See* *Inheritance* pattern.

Types, decomposing patterns, 25–30

**U**

UML (Unified Modeling Language), 46–49.  
*See also* PIN (Pattern Instance Notation);  
 PINboxes.

Unnamed namespaces, 24

Unselfconscious design, 2–3

Used-by relationships, EDPs, 101–102

**V**

*Virtual Method* pattern. *See* *Abstract Interface* pattern.

*Visitor* pattern, 10

Vlissides, John, 2

**W**

Woolf, Bobby, 20, 251

**Z**

Zimmer, Walter, 244