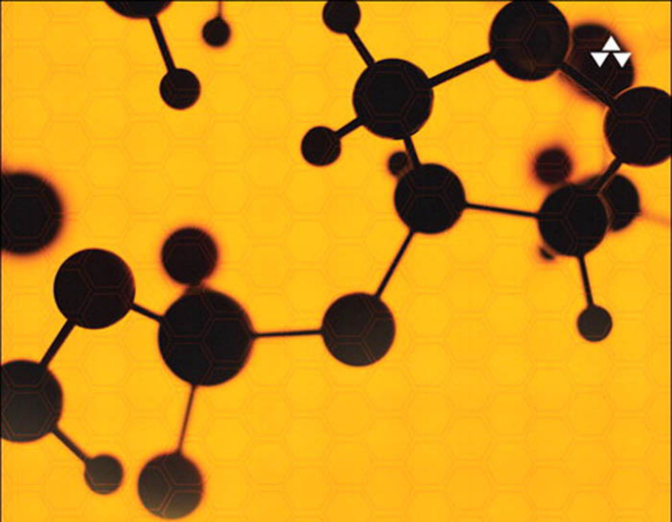

Elemental Design Patterns

JASON McC. SMITH

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

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

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

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

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

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

This page intentionally left blank

Tables

This page intentionally left blank

Listings

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 toolbelt, 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

1 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.[1] *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.[2]

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

N

O

P

T

U

V

W

Z