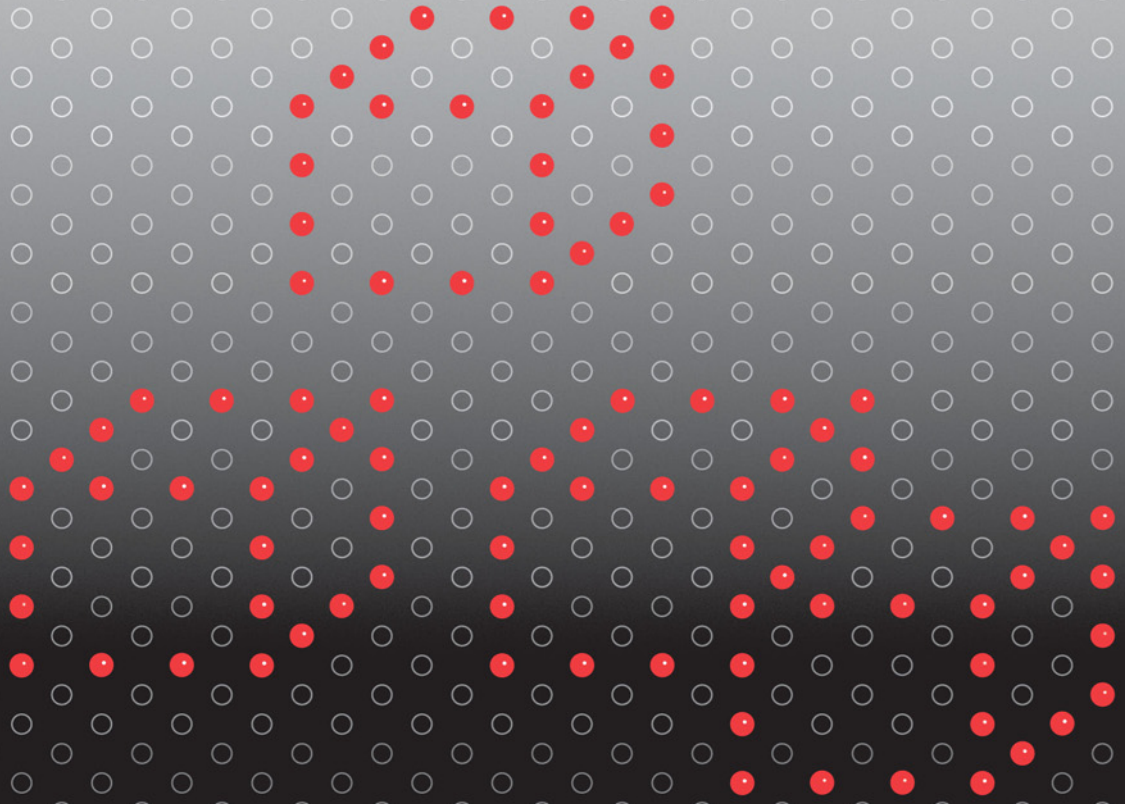
Microsoft

MICROSOFT PROFESSIONAL



Object Thinking

David West

Microsoft

Object Thinking

David West

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data pending.

Printed and bound in the United States of America.

Second Printing: July 2014

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

Acquisitions Editor: Robin Van Steenburgh and Linda Engelman
Project Editor: Denise Bankaitis and Devon Musgrave
Indexer: Shawn Peck

Body Part No. X10-25675

This product is printed digitally on demand.

Table of Contents

Acknowledgments

One name appears on the cover as author of this book, hiding the fact that every book is a collaborative effort involving scores of contributors. Although it is impossible to acknowledge and thank everyone who played a role, I can and must name some individuals. Each is owed my personal thanks, and each stands as a proxy for many others whom I acknowledge in my mind and heart.

Mary—without whom this would never have been done. My muse, my friend, my spouse of twenty-one years.

Maurine, Bob, Sara, Ryan, and Kathleen—my family, whose support was required and freely given.

Kevin—the best programmer with whom I have ever worked. You proved, sometimes after a lot of discussion as to why it was impossible, that my crazy ideas could be implemented.

Tom, Pam, Don, Dion, Julie, Kyle, Dave, Steve, and J.P.—our initial contact was as student and professor, but you became colleagues and friends and represent the hundreds of St. Thomas alumni who helped shape object thinking and then applied it in the real world with notable success.

Tom and Ken—your technical review, insightful comments, correction of errors, and honest advice were invaluable. This would not have been as useful a book without your assistance.

Linda, Devon, Denise, Robin, Shawn, Joel, Sandi, and Elizabeth—the editors and staff at Microsoft Press who smoothly dealt with all the technical aspects of getting a book out the door (including an author with deadline issues). An author could not have found a friendlier, more helpful, or more professional group of editors and craftspeople with whom to work.

This page intentionally left blank

Preface

A Different (and Possibly Controversial) Kind of Software Book

This book will be deliberately different from almost any other object analysis/design, component-based development, software development methodology, or extreme programming (XP) book you may have encountered. It's also likely to be controversial, which is not intended but is, perhaps, inevitable. Several factors contribute to this book's differences:

- The reader will be asked to read and digest a lot of history and philosophy before embarking on the more pragmatic aspects of object thinking.
- The author will unabashedly, adamantly, and consistently advocate behavior as the key concept for discovering, describing, and designing objects and components.
- The centrality of CRC (Class-Responsibility-Collaborator) cards as an object thinking device or tool will likely be viewed as anachronistic and irrelevant since UML (Unified Modeling Language) has achieved the status of de facto standard.
- The apparent indifference, or even antagonism, toward formalism and the formal approaches to software specification that are intrinsic to the behavioral approach will concern some readers, especially those trained in computer science departments at research universities.
- The emphasis on analysis and conceptualization—that is, on thinking—instead of on implementation detail might strike some readers as academic.

It will take the rest of the book to explain why these differences are important and necessary, but the motivation behind them is found in this 1991 quote from Grady Booch:

Let there be no doubt that object-oriented design is fundamentally different than traditional structured design approaches: it requires different ways of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.

"Different ways of thinking" is the key phrase here, and the word *culture* is equally important. The history of objects in software development is characterized by the mistaken notion that the object difference was to be found in a computer language or a software module specification. But objects are fundamentally different because they reflect different ideas—particularly about decomposition—and because they reflect a different set of values and a different worldview (that is, culture) from that of traditional software development. Understanding objects requires understanding the philosophical roots and historical milestones from which objects emerged.

Readers will be expected to review and reflect on two competing philosophical traditions—formalism and hermeneutics-postmodernism—that are responsible for the value systems and ideas leading to the competition and heated disagreements between advocates of software engineering (who value formalism) and advocates of XP/agile methodologies/behavioral objects (who value hermeneutics). A quick look at the history of programming in general will show that XP and agile methodologies are but the latest manifestations in this long-standing philosophical feud, which has also influenced our (mis)understanding of objects and patterns.

Distractions to object thinking, such as which programming language is "best," will be shown to be mostly irrelevant with a brief recap of classical languages claiming to be object-oriented. This historical excursion will also show you how ideas become manifest in tools—in this case programming languages—and how philosophical principles and cultural values shape software development methods and processes.

Metaphor and vocabulary play a major role in shaping object thinking. Metaphors are essential for bridging the familiar and the unfamiliar, and selection of the "right" metaphors is critical for further development of fundamental ideas. Likewise, vocabulary provides us with one of the essential tools for thinking and communicating. It's important to understand why object advocates elected to use a different vocabulary for object thinking and why it's inappropriate to project old vocabulary onto new concepts.

As you might expect by now, the prominent role of formally defined models, methods, and processes will be discounted in this book; their claim to be repositories of objective truth will be challenged. This book will suggest that methods are little more than helpful reminders of "things to think about" and that models are only a form of external short-term memory useful in the context

of a particular group of people at a particular point in time. These biases will be linked to XP/agile approaches.

Note that I'll also provide applications of the ideas this book offers. I'll provide several examples as illustrations of how the ideas exhibit themselves in practice.

However focused on a particular topic each section of this book might be, an overarching bias or perspective colors every discussion: behaviorism. If a single word could be used to capture the "different ways of thinking about decomposition" noted by Booch, it would be *behavior*. The centrality of behavior will be most evident when we consider and compare traditional definitions and models of objects.

A majority of the published books on objects mention using behavior as the criterion for conceptualizing and defining objects. Many claim to present behavioral approaches to object development. However, with one or two exceptions,[1] they provide object definitions and specification models that owe more to traditional concepts of data and function than to behavior. Even when behavior is used to discover candidate objects, that criterion is rapidly set aside in favor of detailed discussions of *data* attributes, member *functions*, and *class/entity* relationships. This book's focus on behavior, on its aspects and implications, is consistent, I believe, with both the origins of object ideas and the design tradition assumed by XP/agile.

Material in this book is presented in a matter-of-fact style, as if everything stated were unequivocally correct. I'll cite alternative ideas and approaches, but I'll make little effort to incorporate those alternatives or to discuss them in detail. This is not because I am unaware of other viewpoints or because I am dismissive of those viewpoints. Alternative ideas are best expressed by their advocates. To the extent that this book is used as a classroom text, discussion of alternative viewpoints and alternative texts is the purview of the instructor.

Paths and Destinations

The destination is easy—to become a better software developer. Or maybe not so easy since "better" requires more than the addition of yet another skill or technique to one's repertoire. Techniques, tools, skills, and facts are part of becoming better but are insufficient in themselves. Integration with existing skills, the transformation of facts into knowledge (even wisdom), and the ability to base one's future actions on principles and ideas instead of rote procedures are all essential to becoming better.

---

1. Wirfs-Brock, Weiner, and Wilkerson's *Objected Oriented Design* offers a behavior-based method, as does Wilkinson's *Using CRC Cards*.

This book is intended to offer one path to becoming a better developer. Specifically, a better agile developer; and more specifically yet, a better extreme programmer. By better, I mean more like the acknowledged masters of agile development and extreme programming: Ward Cunningham, Kent Beck, Ron Jeffries, Alistair Cockburn, Bob Martin, Ken Auer … (Please fill in the rest of the top 20 to reflect your own heroes.)

These people are not masters merely because they practice certain techniques or exhibit certain skills. They embody, they live, they exude shared values and principles. They share similar experiences and have learned common lessons from those experiences. They share a worldview. Given a common anthropological definition of culture—"shared, socially learned knowledge and patterns of behavior"[2]—it is reasonable to assert that the agile and extreme programming masters constitute a culture—a subculture, actually—of software developers. If one aspires to be like them, one must become a member of that culture.

The process of learning a culture—enculturation—is partly explicit but mostly implicit. The explicit part can be put into books and taught in seminars or classrooms. Most of culture is acquired by a process of absorption—by living and practicing the culture with those who already share it. No book, including this one, can replace the need to live a culture. But it is possible to use a book as a means of "sensitizing," of preparing, a person for enculturation—shortening the time required to understand and begin integrating lived experiences. That is the modest goal of this author for this book.

Who Should Read This Book

A conscious effort has been made to make this book useful to a wide audience that includes professional developers and postsecondary students as well as anyone with an interest in understanding object-oriented thinking. It's assumed that the average reader will have some degree of familiarity with software development, either coursework or experience. It's further assumed that the reader will have been taught or will have experience using the vocabulary, models, and methods prevalent in mainstream software development. This book focuses on presenting objects and XP and not on providing background and details of mainstream development.

---

2. Peoples, James, and Garrick Bailey. *Humanity: an Introduction to Cultural Anthropology*. Belmont, CA: Wadsworth/Thompson Learning, 2000.

The organization of the book, front-loading the philosophical and historical material, might present some problems for the most pragmatically oriented reader, the one looking for technique and heuristics to apply immediately. The introduction attempts to make a case for reading the book in the order presented, but it's OK to skip ahead and come back to the early chapters to understand the ideas behind the technique.

Three caveats help to further define the audience for this book:

- This is not a programming text. Some limited examples of pseudocode (usually having the flavor of Smalltalk, the language most familiar to the author) are presented when they can illuminate a concept or principle of development.
- The book, however, is expressly intended for programmers, especially those using Java and C++. It's hoped that this book will facilitate their work, not by providing tricks and compiler insights but by providing conceptual foundations. Languages such as C++ and Java require significant programmer discipline if they are to be used to create *true* objects and object applications. That discipline must, in turn, be grounded in the kind of thinking presented in this book.
- Although this book stresses the philosophy and history of objects/components, the author's overarching goal is assisting people in the development of pragmatic skills.

> **Note** If this text is used in an academic course (undergraduate or graduate), roughly 40 percent of the available class time should be devoted to workshop activities. Software development of any kind is learned through experience, but objects, because they are new and different, require even greater amounts of reflection and practice. Another characteristic of an ideal course is interaction and discussion of multiple viewpoints. This book is best used in conjunction with other method books (particularly any of the excellent books on UML or RUP) that present alternative viewpoints and, of course, with at least one of the XP or agile texts, which will add depth to what is presented here.

Portions of the historical and philosophical material in this book will be familiar to those readers with their own extensive personal experience as practitioners. To them, the material might appear to be rehashed old arguments. But

most readers will not share either that experience or that awareness, and they need to know, explicitly, how those ideas have shaped the profession as we see it today. I hope that even the most experienced practitioners will see some new insights or have old insights brought to the forefront of their minds when reading this material.

One final note. Readers of this book will tend to be professional software developers or those aspiring to become professionals. Both groups share a character trait—an eagerness to build, to practice. They are ready to "just do it."

To them is directed the final message of this introductory chapter: please be patient.

As a profession, we also tend to be abysmally ignorant of our own history. In the words of the oft quoted, misquoted, and paraphrased philosopher, George Santayana, "Those who cannot remember the past are condemned to repeat it." In computing, thanks to Moore's Law, our repetition is disguised a bit by scale—mainframe mistakes replicated on desktops, then notebooks, then PDAs and phones, and then molecular computers.

As a profession, we tend to ignore the various influences—such as culture, philosophy, psychology, economics, and sheer chance—that have shaped the practice of software development.

It is my belief that we cannot improve our intrinsic abilities as object and agile software developers without an examination, however brief, of the historical and philosophical presuppositions behind our profession. We should not, and cannot if we want them to truly understand, bring new members into our profession without providing them with more than tools, methods, and processes that they use by rote.

How This Book Is Organized

Readers are encouraged to proceed through the book in sequence, to accept that each chapter lays a foundation for the one that follows. Overlaying the order are several implicit logical divisions, including the following:

- The introduction and Chapter 1, "Object Thinking," advance arguments for why understanding the background and history of ideas is a necessary step in the successful application of those ideas. They also argue that both objects and extreme programming (agile methods) share common foundations.
- Chapter 2, "Philosophical Context," and Chapter 3, "From Philosophy to Culture," provide a foundational context, partly based in philosophy and partly in history.

- Chapter 4, "Metaphor: Bridge to the Unfamiliar," introduces key ideas and meta-ideas (metaphors as ideas about how to explain ideas, simply put).
- Chapter 5, "Vocabulary: Words to Think With," introduces vocabulary and explains why object thinking requires a different vocabulary for things that appear to be familiar but that use old labels.
- Chapter 6, "Method, Process, and Models," revisits commonalities between object and agile ideas and the relationship of those ideas to the notion of method and process.
- Chapter 7, "Discovery," Chapter 8, "Thinking Toward Design," and Chapter 9, "All the World's a Stage," apply the ideas of previous chapters and provide examples of object thinking in action.
- Chapter 10, "Wrapping Up," adds a coda with short explorations of how object thinking can be extended and coordinated with unavoidable nonobject worlds, and what the ultimate outcome of object thinking might be.

I encourage you to engage the book in the order presented, but feel free to skip Chapter 1 if you're prepared to take on faith the assertions made about objects and object thinking in subsequent chapters.

Many readers will be a bit anxious to "get to the good stuff," to see how the book's ideas are applied or how they manifest themselves in practice (which is shown in Chapters 7 through 9). While I firmly believe that the book's initial chapters—as far afield as they might appear to be—need to be covered first, I also recognize the need to at least foreshadow application. To this end, I offer a continuing sidebar—the "Forward Thinking" sidebar—in these initial chapters. This sidebar presents a sample application in bits and pieces as a means of illustrating important ideas while satisfying the craving for pragmatic examples.

Introduction

The time: 1968. A software crisis has been declared. Part of the crisis derives from the fact that more software is needed than there are developers to produce it. The other part of the crisis is the abysmal record of development efforts. More than half of the projects initiated are canceled, and less than 20 percent of projects are successfully completed, meaning that the time and cost overruns associated with those projects were less than 100 percent and the software was actually used to support the business.

The time: 2003. We have a lot of developers—part of the software crisis has passed for the moment. Many of them are located in other countries because too many managers seem to believe that developers are developers and therefore go for greater numbers at lesser cost.

The skills, abilities, attitudes, and aptitudes of the development community are, unfortunately, suspect. The "development community" means the *entire* development community—from programmers to senior managers—in the United States and the rest of the world. It is still the case that almost half of all projects initiated are not completed. Those that are completed almost always incur significant cost overruns. Quality—the lack thereof—is still a major issue. Bloated, inefficient, bug-ridden, user-unfriendly, and marginally useful—these are still common adjectives used to describe software.

The lack of significant improvement is not for want of trying. Numerous proposals have been advanced to improve software development.

Software engineering provides an umbrella label for a spectrum of improvement efforts ranging from structured programming to formal (mathematical and logical) foundations for program specification to UML (unified modeling language) to ISO 9000 certification and CMM (capability maturity model) formalizations of the development process.

Sharing the goal of software improvement but rejecting the assumptions implicit in software engineering, several alternative development approaches have been championed. Creativity and art have been suggested as better metaphors than engineering. Software craftsmanship[1] is the current incarnation of this effort. Iterative development has as long a history as the linear-phased development at the heart of software engineering.

---

1. McBreen, Peter. *Software Craftsmanship: The New Imperative.* Addison-Wesley, 2001.

Extreme programming (XP) and agile software development represent a contemporary attempt to redefine and improve software and the practice by which that software comes into existence. XP/agile represents an alternative to software engineering. Further, XP/agile challenges the assumptions, the core values, and the worldview upon which software engineering is predicated. It is not surprising, therefore, that those with a vested interest in software engineering are trying very hard to marginalize ("it's OK for small projects by small teams that are not engaged in mission-critical projects"), co-opt ("XP is just a subset of RUP[2]—with a different vocabulary"), or dismiss ("XP is just a tantrum staged by a few petulant out-of-work Smalltalk programmers angry at the demise of their favorite programming language").

This book is motivated by a set of curious questions; it is based on the belief that "better people" are absolutely essential and that they can be nurtured; it is grounded in the conviction that object-oriented ideas and principles are poorly understood; and it is premised on the belief that XP (and other agile methods) and object thinking are inextricably entwined—each requires the other if they are to be successful.

Curiosities

Consider the following curious questions: Why is it that iterative development has been acknowledged—even by those proposing sequential (waterfall) development—as the "right" way to produce software, and yet

- XP is seen as a "novel" and even "radical" approach?
- So few, except those who have attempted to install an agile approach the past two or three years, "officially" use iterative development?
- There is so much resistance—by managers and professional developers—to doing things the "right" (iterative) way?

2. RUP—Rational Unified Process, an attempt to standardize and formally define the software development process. Created and advocated by the same individuals behind UML (unified modeling language), most notably Grady Booch, Ivar Jacobson, and James Rumbaugh.

- Discussions between proponents and opponents of agile approaches are so heated and frequently emotional when both sides seem to agree, on the surface, on so many things?
- "Better people" has been recognized as the most promising silver bullet for addressing the software crisis, and yet almost all of our energy has been spent on creating better tools, methods, and processes instead of better people?
- Every effort to advance nonformal, iterative, artistic, and humane ways to develop software seem to be resisted and then co-opted and debased by formal software engineering?
- So few developers seem to be able to adopt and practice innovations such as objects and agile methods in a manner consistent with the intent and example of those who first advanced the innovation? (Don't believe this is a fair statement? Then why, two to four years after XP and agile were introduced, is that community spending so much time and effort wrestling with questions of certification? Why does Alan Kay say the object revolution has yet to occur?)

The long answer to these and similar questions is this book. The short answer, and hopefully part of your motivation for reading this book, is that software developers tend to be so focused on what and how that they forget to explore questions of why.

The "People Issue"

Fact 1: The most important factor in software work is not the tools and techniques used by the programmers but rather the quality of the programmers themselves.

Fact 2: The best programmers are up to 28 times better than the worst programmers, according to "individual differences" research. Given that their pay is never commensurate, they are the biggest bargains in the software field.

—Robert L. Glass[3]

3. Glass, Robert L., *Facts and Fallacies of Software Engineering*. Boston: Addison-Wesley, 2003.

In his discussion of the foregoing facts, Glass notes that we have been aware of these human differences since 1968:[4]

Nearly everyone agrees, at a superficial level, that people trump tools, techniques, and process. And yet we keep behaving as if it were not true. Perhaps it's because people are a harder problem to address than tools, techniques, and process.

We in the software field, all of us technologists at heart, would prefer to invent new technologies to make our jobs easier. Even if we know, deep down inside, that the people issue is a more important one to work.

—Glass, 2003

This book is an attempt to address the "people issue." Specifically, it is an attempt to help developers improve their intrinsic abilities.

The Need for Better Developers

For decades, the profession of software development has relied on an implicit promise: "Better tools, methods, and processes will allow you to create superior software using average human resources." Managers have been sold the idea that their jobs will be made easier and less risky by adoption of strict formal methods and processes—a kind of intellectual Taylorism. (Taylor, Gilbreth, and others advanced the concept of scientific management, time and motion studies, and the modern assembly line. Their distrust of human workers led them to the idea that imposed process and method could compensate for weaknesses they felt to be innate in workers. The same attitude and philosophy are assumed in the field of software engineering.) For the most part, these efforts have failed miserably.

Instances of success exist, of course.[5] You can find cases in which project A using method X did, in fact, achieve notable success, but industry statistics as a whole have failed to improve much since 1968, when software engineering and scientific management were introduced as means for resolving the software crisis. Abandoned projects, cost/time overruns, and bloated, buggy software still dominate the landscape.

---

4. Sackman, H., W.I. Erikson, and E.E. Grant. "Exploratory Experimental Studies Comparing Online and Offline Programming Performances." *Communications of the ACM*, January 1968.

5. Some advocates of software engineering and process improvement will make claims of major successes (SEI Report CMU/SEI-2001-SR-014).

Even the most ardent advocates of software engineering (for example, Dykstra, Boehm, and Parnas), as well as those most responsible for popularizing software engineering among the corporate masses (Yourdon and Martin), recognized the limits of formal approaches. In "A Rational Design Process: How and Why to Fake It,"[6] Parnas acknowledged the fact that highly skilled developers did not "do" development the way that so-called rational methods suggested they should. Martin suggested that the best way to obtain high-quality software—software that met real business needs on time and within budget—was the use of special "SWAT" teams comprising highly skilled and greatly rewarded developers doing their work with minimal managerial intervention.

The only consistently reliable approach to software development is, simply, good people. So why have so much attention and effort been devoted to process and method? There have been at least three contributing reasons:

- A widespread belief, partially justified, that not enough good people were available to accomplish the amount of development that needed to be done.
- An unspoken but just as widely held belief that really good developers were not to be trusted—they could not be "managed," they all seemed to be "flaky" to some degree, and they did not exhibit the loyalty to company and paycheck of "normal" employees. Really "good" developers tended to be "artists," and art was (is) not a good word in the context of software development.
- A suspicion, probably justified, that we did not (do not) know how to "upgrade" average developers to superior developers except by giving them lots of experience and hoping for the best.

It is no longer possible to believe that either method or process (or both together) is an adequate substitute for better people. There is a resurgence of interest—spurred most recently by XP and the core practices in other agile methods—in how to improve the human developer. This interest takes many forms, ranging from XP itself to redefinition of software development as a craft (McBreen[7]) and a calling (West[8], Denning[9]) instead of a career to software apprenticeship (Auer[10]). Why do we need better developers? Because increasing

6. Parnas, David Lorge, and Paul C. Clements. "A Rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineering*. Vol. SE-12, No. 2, February 1986.
7. McBreen, Peter. *Software Craftsmanship: The New Imperative*. Boston: Addison-Wesley, 2001.
8. West, David. "Enculturating Extreme Programmers," Proceedings of XP Universe, Chapel Hill, NC., July 2001. "Educating Xgilistas," Educator's Forum-XP/Agile Universe, Edmonton, Canada, July 2004.
9. Denning, Peter J. "Career Redux," Communications of the ACM. September 2002 / Vol 45 No. 9.
10. Ken Auer, *www.rolemodelsoft.com*.

the supply of highly skilled people—rather than only adhering to particular methods and processes—is the only way to resolve the software crisis.

> **Note** Extreme programming and agile methods derive from the actual practice of software development rather than academic theory. The critical difference between XP/agile and traditional methods is their focus on attitudes, behaviors, culture, and adaptive heuristics instead of formal technique and theory. If the "method" label is to be attached to XP/agile, it should be in terms of a method for producing better developers rather than a method for producing better software. Better software comes from, and only from, better people.

Producing Better Developers

What makes a better developer? The traditional answer is experience. Most textbooks on software engineering and formal process contain a caveat to the effect that extensive real-world experience is required before the tools and techniques provided by method/process can be effectively utilized. In this context, experience is just a code word for those aspects of development—philosophy, attitude, practices, mistakes, and even emotions—that cannot be reduced to syntactic representation and cookbook formulation in a textbook.

Today's practitioners, and some theorists, are intensely interested in teasing apart and understanding the complex elements behind the label "experience." There is great desire to understand what developers actually "do" as opposed to some a priori theory of what is appropriate for them to do. Models are seen as communication devices, unique to a community of human communicators, instead of "vessels of objective truth" with a value that transcends the context of their immediate use. Development is seen as a human activity—hence a fallible activity—that must accommodate and ameliorate imperfection and mistakes rather than seek to eliminate them. Communalism is valued over individualism. Systems are seen as complex instead of just complicated, necessitating a different approach and different insights than were required when software developers merely produced artifacts that met specification.

All of this intense interest is producing results. Although the total picture is still emerging, some facets of "the better developer" are coming into focus.

XP provides a foundation by identifying a set of discrete practices that can actually be practiced. Any developer can actually do these practices and, simply by doing them, become a better practitioner. They offer no grand theory, nor

are they derived from any theory. The justification for the XP approach is based on two simple empirical observations: "We have seen master developers do these things" and "We have seen less proficient developers do these things and become better." Although XP does make claims to improve both product and process, these are side effects—one is tempted to say mere side effects—of the improvement in the human developers.

XP/agile approaches provide a solid foundation, but just a foundation. What "Xgilistas"[11] *do* is but part of what makes them master developers. What they think and how they think are critically important as well. XP relies on maximized communication and storytelling as a means for enculturating new developers in appropriate ways of thinking. Thinking includes a value system, a history, a worldview, a set of ideas, and a context. And XP encompasses an oral tradition that has not, as yet, been reduced to ink and paper (and maybe cannot be so reduced). Aspiring Xgilistas must become conversant with all of this before they can attain true "master" status.

One component of the oral tradition, of the history, and of the common worldview is the use of object-oriented approaches to design and programming. Unfortunately, this is seldom made explicit in the XP literature. The terms *object* and *object-oriented* do not appear in any of the first five books in the Addison-Wesley XP series—except once, and that occasion points to an incorrect page in the text. However, object vocabulary and concepts are abundantly evident. This discrepancy merely confirms that object thinking is presupposed by those advocating XP. The primary goal of this book is to provide one small contribution to help those following the Xgilista path—specifically, a contribution in the area of *object thinking*.

Object Thinking

Thirty plus years have passed since Alan Kay coined the term *object-oriented*. Almost all contemporary software developers describe their work using object vocabulary and use languages and specification tools that lay claim to the object label. The ubiquity of object terminology does not mean, however, that everyone has mastered object thinking. Nor does the popularity of Java. Nor does the de facto standardization of object modeling embodied in UML. A prediction made by T. Rentsch (cited by Grady Booch in 1991[12]) remains an accurate description of today's development and developers:

---

11. A neologism—useful for labeling those that embody one or more of the approaches that fall under the "agile" label.

12. Booch, Grady. *Object-Oriented Analysis and Design with Applications,* Second Edition. Boston: Addison-Wesley, 1993.

My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.

In fact, the situation may be worse than Rentsch predicted. An argument can be made that the contemporary mainstream understanding of objects is but a pale shadow of the original idea. Further, it can be argued that the mainstream understanding of objects is, in practice, antithetical to the original intent.

Clearly the behavioral approach to understanding objects has all but disappeared. (Yes, UML does allow for a "responsibility" segment in its class diagram and description, but this hardly offsets the dominant trend in UML to treat objects as if they were "animated data entities" or "miniature COBOL programs.") This fact is important because two of the leading advocates behind XP were also the leading advocates of "behavioral objects." Kent Beck and Ward Cunningham invented the CRC card approach to finding and defining objects—the most popular of the "behavioral methods." Others deeply involved in the Agile Alliance were also identified with "object behavioralism" and with Smalltalk, the programming language that came closest to embodying behavioral objects.

It isn't unreasonable to assume that the behavioral approach to understanding objects dominates the "object thinking" of many of the best XP practitioners, in part because it was almost necessarily part of the oral tradition passed on by Kent Beck as he initiated others into the XP culture.

Note Certain aspects of XP practice are simply incarnations of earlier practices. XP "stories," for example, which deal with a system function actualized by a group of objects, are identical to the scenarios talked about in conjunction with CRC cards. Stories that are factored to their simplest form are nothing more than responsibilities. The only real difference between stories and scenarios/behaviors is the insistence that stories are always, and only, told by the domain expert (the customer) and always depict interactions in the problem domain. This was also true of good scenarios, but in the object literature scenarios tended to reflect implementation more than they did problem description.

It's also reasonable to assume that behavioral object thinking is only implicit in the XP/agile culture because so few books or texts were ever written in support of this approach. Neither Beck nor Cunningham ever wrote such a book. Rebecca Wirfs-Brock didn't update her 1991 book describing behavior-based object design until this year. Other efforts in this area (for example, Nancy Wilkerson's book on CRC cards) were not widely known and have not been updated.

This is particularly unfortunate because the CRC card "method" as described in the early 1990s did not incorporate all aspects of object thinking. In fact, I believe that object thinking transcends the notion of method just as it transcended programming languages. (You can do good object programming in almost any language, even though some languages offer you more support than others.)

Object thinking requires more than an understanding of CRC cards as presented circa 1990. It also requires understanding some of the history and some of the philosophical presuppositions behind object behavioralism, CRC cards, and languages such as Smalltalk. It requires an understanding of the metaphors that assist in good object thinking and an extension of the CRC card metaphor, in particular, to include more than object identification and responsibility assignment.

It is my hope that this book will promote such an understanding by capturing at least part of the oral tradition of behavioral objects and making it explicit.

XP and Object Thinking

This book is based on the following beliefs:

- Agility, especially in the form of extreme programming, is essential if the software development profession, and industry, are to improve themselves.
- XP offers a way to create "better developers" to improve the intrinsic abilities of human beings to develop software for use by other human beings—a way to produce "master" developers.
- XP cannot be understood, and those practicing XP will not realize the full potential of the approach, until they understand object thinking and the shared historical and philosophical roots of both object thinking and XP core values and practices.
- In particular, programmers and technical developers will fail to realize the full potential of XP-based development without a thorough understanding of object orientation—the "object thinking" promised by the title of this book.

3

From Philosophy to Culture

Philosophy provides roots, determines some values, and affects the design of some tools, but the fullness of the object thinking difference must be understood in terms of a broader context—as a culture. The cultural perspective suggests the need to look for the shared, socially learned knowledge (norms, values, worldviews) and patterns of behavior (individual actions and organizational relationships) that characterize a group of people. The philosophical positions discussed in the preceding chapter are part of the cultural knowledge shared by object thinkers as members of an object culture.

Robert Glass used the metaphor of culture to explain differences in how different groups of people conceive of and develop software and how the results of their work are evaluated. His contrast of Roman and Greek cultures directly parallels the contrast made in the preceding chapter between formalist and hermeneutic philosophies. The Greek culture described by Glass is a close match to the XP culture[1] and the object culture we will explore in this chapter.

1. West, David. "Enculturating Extreme Programmers," Proceedings XP Universe. Chapel Hill, N.C., 2001.

Greek and Roman

Robert Glass makes the argument for two cultures within the realm of software development, two cultures that frequently find themselves in conflict based on cultural values. He uses an analogy with Greek and Roman culture to illustrate the differences as follows:

In ancient Greece, an individual would act as his own agent in his own behalf, or combine with other people to act together as a team. In a Greek work environment, you bring your tools to work with you, you do your stuff, and then you pack up your tools and take them home. You are an individual—an independent contractor. You are not owned body and mind. You are merely providing a service for compensation.

In Rome, one's first duty was to the group, clan, class, or faction upon which one depended for status. Known as gravitas, this meant sacrificing oneself for the good of the organization, and giving up one's individuality and identifying closely with the group. In a Roman environment you go to work, the company hands you your tools, and then it holds you and your mind hostage until you sever your relationship with the organization. You are not an individual: you are owned by the organization body and mind, twenty-four hours a day. There are substantial rewards for this, however. The organization provides you with security, money, and power.

Glass is particularly interested in the degree to which the two cultures support creativity and asserts that the Roman culture is likely to take the creativity, passion, and magic out of the work of software development. He further notes that Roman culture will emphasize up-front planning, control, formal procedures as a means of control, and maximum documentation and will value logical, analytical thinking above empirical and inductive thinking.

Even a cursory evaluation of XP values and practices reveals their incompatibility with Roman thinking. When objects were first introduced, they too reflected a Greek and not a Roman culture. Smalltalk was motivated by a need to empower people, to make interaction with a computer fun and creative. Exploratory development, a kind of rapid prototyping, was seen as the proper way to develop new software—as opposed to the notion of up-front, detailed design and rote implementation favored by the (Roman) structured development culture.

Our exploration of the object culture begins with a rough enumeration of some groups likely to be included in this culture. The original proponents of object ideas (the original SIMULA team, the Smalltalk team), advocates of behavior-based object methods, and the first XP and agile practitioners are likely candidates, and of course, any who recognize themselves as members of the "Greek" culture described by Glass.

Absent a full ethnography, a cursory look at the object culture reveals some important traits and characteristics. Specifically:

- A commitment to *disciplined informality* rather than defined formality
- Advocacy of a local rather than global focus
- Production of minimum rather than maximum levels of design and process documentation
- Collaborative rather than imperial management style
- Commitment to design based on coordination and cooperation rather than control
- Practitioners of rapid prototyping instead of structured development
- Valuing the creative over the systematic
- Driven by internal capabilities instead of conforming to external procedures

Cultures will usually have an origin myth, various heroes and heroines, and stories about great deeds and important artifacts, as well as a set of core beliefs and values. All of these are evident in the object culture and someday will be captured in a definitive ethnography. It is not my intent to elaborate that culture here, merely to draw the reader's attention to the fact that such a culture exists.

Being aware of object culture is valuable for object thinkers in four ways. First, it provides insight into the dynamics of interaction (or lack of it) between objectivists and traditionalists. Often the only way to understand the mutual miscommunications and the emotional antipathy of the two groups comes from understanding the underlying cultural conflict.

Second, and most important, it reminds the aspiring object thinker that he or she is engaged in a process of enculturation, a much more involved endeavor than learning a few new ideas and adopting a couple of alternative practices. Most of the material in the remainder of this book cannot be fully understood without relating it to object culture in all its aspects.

Third, it suggests a way to know you have mastered object thinking. When all of your actions, in familiar and in novel circumstances, reflect "the right thing" without the intervention of conscious thought, you are an object thinker.

Fourth, it reminds the object thinker that culture is not an individual thing; it is rooted in community. To change a culture, you must change individuals and the way that individuals interact and make commitments to one another. Culture is shared.

Subsequent chapters will deal with object thinking specifics, all of which are intimately related to a set of *first principles*, or presuppositions reflective of the object culture as a whole. The four principles introduced here are frequently stated, stated in a manner that implies that the value of the principle is obvious. Just as a member of any culture makes assertions that are indeed obvious—to any other member of that culture.

Four Presuppositions

To those already part of the object culture, the following statements are obvious ("they go without saying") and obviously necessary as prerequisites to object thinking:

- Everything is an object.
- Simulation of a problem domain drives object discovery and definition.
- Objects must be composable.
- Distributed cooperation and communication must replace hierarchical centralized control as an organizational paradigm.

For those just joining the object culture, each of these key points will need to be developed and explained. For those opposed to the object culture, these same presuppositions will be major points of contention.

One: Everything is an object.

This assertion has two important aspects. One is essentially a claim to the effect that the object concept has a kind of primal status—a single criterion against which everything else is measured. Another way of looking at this claim would be to think of an object as the equivalent of the quanta from which the universe is constructed. The implication of this aspect of everything-is-an-object suggests that any decomposition, however complicated the domain, will result in the identification of a relatively few kinds of objects and only objects.

There will be nothing "left over" that is not an object. For example:

- *Relationships*, which are traditionally conceived as an association among objects that is modeled and implemented in a different way than an object—would themselves become just another kind of object, with their own responsibilities and capabilities.

- *Data* traditionally is seen as a kind of "passive something" fundamentally different from "active and animated things" such as procedures. Something as simple as the character *D* is an object—not an element of data—that exhibits behavior just as does any other object. Whatever manipulations and transformations are required of an object, even a character, are realized by that object itself instead of some other kind of thing (a procedure) acting upon that object. The commonsense notion of data is preserved because some objects have as their primary, but not exclusive, responsibility the representation to human observers of some bit of information.

- *Procedures* as a separate kind of thing are also subsumed as ordinary objects. We can think of two kinds of procedure: a script that allows a group of objects to interact in a prescribed fashion and the "vital force" that actually animates the object and enables it to exhibit its behaviors. A script is nothing more than an organized collection of messages, and both the collection and the message are nothing more than ordinary objects. The vital force is nothing more than a flow of electrons through a set of circuits—something that is arguably apart from the conceptual understanding of an object, just as the soul is deemed to be different from but essential to the animation of a human being.

Equating, even metaphorically, a procedure to a soul will strike most readers as a bit absurd, but there is a good reason for the dramatic overstatement. It sometimes takes a shock or an absurdity to provide a mental pause of sufficient length that a new idea can penetrate old thinking habits. This is especially true when it comes to thinking about programming, wherein the metaphysical reality of two distinct things—data and procedures—is so ingrained it is difficult to transcend. So difficult, in fact, that most of those attempting object development fail to recognize the degree to which they continue to apply old thinking in new contexts.

Take object programming, for example—using Smalltalk as an example merely because it claims to be a pure object language. Tutorials from Digitalk's Smalltalk manuals illustrate how programmers perpetuate the notion that some things "do" and others are "done to."

The code in **Listing One—Pascal** is a Pascal program to count unique occurrences of letters in a string entered by a user via a simple dialog box. (Pascal was designed to teach and enforce that algorithms [active procedures] plus [passive] data structures = program mode of thinking.)

Listing Two—Naive Smalltalk shows an equivalent Smalltalk program as it might be written by a novice still steeped in the algorithms plus data structures mode of programming. Both programs contain examples of explicit control and overt looping constructs. The Pascal program also has typed variables—an implicit nod to the need for control over the potential corruption of passive data.

Listing One—Pascal

```
program frequency;
      const
        size 80;
      var
        s: string[size];
        i: integer;
        c: character;
        f: array[1..26] of integer;
        k: integer;
     begin
        writeln('enter line');
        readln(s);
        for i := 1 to 26 do f[i] := 0;
        for i :=  1 to size do
            begin
            c := asLowerCase(s[i]);
            if isLetter(c) then
                begin
                k := ord(c) - ord('a') + 1;
                f[k] := f[k] + 1
                end
           end;
        for i := 1 to 26 do
           write(f[i], ' ')
     end.
```

There is some evidence of object thinking in Listing Two—mostly conventions or idioms enforced by the syntax of the Smalltalk language—the use of the *Prompter* object, the control loops initiated by integer objects receiving messages, discovery of the size of the string being manipulated by asking the string for its size, and so forth.

Listing Two—Naive Smalltalk

```
| s c f k |
f := Array new: 26.
s := Prompter prompt: 'enter line' default: ' '.
1 to: 26 do: [:i | f at: i put: 0].
1 to: s size do: [
      :I | c := (s at: i) asLowerCase.
      c isLetter ifTrue: [
         k := c asciiValue - $a asciiValue + 1.
         f at: k put: (f at: k) + 1.
                          ].
                 ].
 ^ f
```

A programmer better versed in object thinking (and of course, the class library included in the Smalltalk programming environment) starts to utilize the

innate abilities of objects, including *data* objects (the string entered by the user and character objects), resulting in a program significantly reduced in size and complexity, as illustrated in **Listing Three—Appropriate Smalltalk**.

Listing Three—Appropriate Smalltalk

```
| s f |
s := Prompter prompt: ' enter line ' default: ' '.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [f add: c asLowerCase]].
^ f.
```

Types, as implied earlier, create a different kind of thing than an object. Types are similar to classes in one sense, but classes are also objects and types are not. This distinction is most evident when variables are created. If variables are typed, they are no longer just a place where an object resides. Typing a variable is a nonobject way to prevent all but a certain kind of object from taking up residence in a named location. Many people have advanced arguments in favor of typing, but none of those arguments directly challenges the everything-is-an-object premise. The arguments in favor of types are orthogonal to the arguments in favor of treating everything as an object. (See note.)

> **Note** Programs are written by human beings, and human beings make mistakes. One response to this truism is to assume that the quantity of mistakes is both high and essentially constant, which mandates the existence and use of error detection and prevention mechanisms—such as typing. Mechanisms, such as typing, are always constrictive, so much so that every typed language I know of allows ways to escape the confines of strict typing—casting, for example—which reintroduce the potential for the errors that typing was intended to prevent. An alternative response, one consistent with the ideas and ideals of object thinking, is to reduce the programmer's proclivity for making errors by teaching the programmer the precepts of simplicity and testing. Most of the time, data moves about a program with little chance of error arising from the wrong kind of data being in the wrong place at the wrong time. (User input is the obvious major exception.) If your thinking about objects and object communication reveals the potential for a type error, you should create a test for such errors and include an explicit check in your code at that point. ("Element of data occupying variable *X*, what class are you an instance of?") Since everything is an object, your element of data is an object quite capable of telling you its class. You can have all the benefits of typing without the constraints and the complications arising from escape valves such as casting.

The everything-is-an-object principle applies to the world, the problem domain, just as it applies to design and programming. David Taylor[2] and Ivar Jacobson[3] use objects as an appropriate design element for engineering, or reengineering, businesses and organizations. (See the sidebar, "David A. Taylor and Convergent Engineering.")

David A. Taylor and Convergent Engineering

Traditional modeling of businesses and organizations is flawed according to Taylor because of the lack of consistency among the set of models utilized. For example, neither a financial model nor a data model captures the cost of a bit of information, and inconsistency in design philosophy prevents the two models from collectively revealing such costs—they cannot be coordinated.

In his book *Business Engineering with Object Technology* (John Wiley and Sons, 1995), Taylor suggests creating a single object model incorporating everything necessary to produce traditional financial, simulation, process, data, and workflow models as *views* of the unifying object model. His process for accomplishing this goal is *convergent engineering*, and it, in turn, is based on a behavioral, CRC (Class, Responsibility, Collaborator) card, approach to object discovery and specification.

In addition to describing how to conceptualize objects and classes, Taylor describes a process for discovery and specification leading to the creation of the organizational object model. That model identifies all the objects in an organization and how they interact—not just the ones that will eventually be implemented as software. He also provides a framework for business objects that illustrates the power of object thinking in generating simple but powerful objects. His framework defines four classes (*Business Elements*, *Organizations*, *Processes*, and *Resources*), describes the behaviors of each, how those behaviors contribute to the generation of the five standard types of business model, how they can be customized, and how their interoperation can be optimized to reengineer the organization as a whole.

2. Taylor, David. *Business Engineering with Object Technology*. John Wiley & Sons, 1995.

3. Jacobson, Ivar. *The Object Advantage: Business Process Reengineering with Object Technology*. ACM Press. Reading, MA: Addison-Wesley. 1994.

The programming example shown earlier illustrates one dimension of treating everything as an object. Applying the everything-is-an-object principle to the world—finding and specifying objects that are not going to be implemented in program code or software—can be illustrated by considering a *Human* object. Objects, as we will discuss in detail later, are defined in terms of their behaviors. A behavior can be thought of as a service to be provided to other objects upon request.

What services do humans provide other objects? For many, this is a surprising question because human beings are not "implemented" by developers and are therefore considered outside the scope of the system. But it is a fair question and should result in a list of responsibilities similar to the following:

- Provides information
- Indicates a decision
- Provides confirmation
- Makes a selection

The utility of having a *Human* object becomes evident in the simplification of interface designs. Acknowledging the existence of *Human* objects allows the user interface to reflect the needs of software objects for services from *Human* objects. This simple change of perspective—arising from application of the everything-is-an-object principle—can simplify the design of other objects typically used in user-interface construction.

Additional implications of the everything-is-an-object premise will be seen throughout the remainder of this book.

Two: Simulation of a problem domain drives object discovery and definition.

Decomposition—breaking a large thing up into smaller, more easily understood things—is necessary before we can solve most of the problems we encounter as software developers. There are different approaches to decomposition. For example, find the data and data structures, find the processing steps, and find the objects. In object thinking, the key to finding the objects is simulation. The advocacy of simulation for object discovery has four primary roots:

- The system description language philosophy behind SIMULA, as discussed in the preceding chapter.
- Alan Kay's ideas about user illusions and objects as reflections of expectations based on an understanding of how objects behave in a domain.

- David Parnas's arguments in favor of a "design decision hiding" approach to decomposition—partitioning the problem space and not the solution space as did functional decomposition approaches—as discussed in the preceding chapter.
- Christopher Alexander's[4] ideas about design as the resolution of forces in a problem space and his subsequent work on patterns that underlie the organization of a problem space and provide insights into good design. These will be elaborated later in this book in the discussion of patterns and pattern languages as an aspect of object thinking.

Proper decomposition has been seen as *the* critical factor in design from very early times. This quotation from Plato (which you might recall from Chapter 1) is illustrative.

[First,] perceiving and bringing together under one Idea the scattered particulars, so that one makes clear the thing which he wishes to do... [Second,] the separation of the Idea into classes, by dividing it where the natural joints are, and not trying to break any part, after the manner of a bad carver... I love these processes of division and bringing together, and if I think any other man is able to see things that can naturally be collected into one and divided into many, him I will follow as if he were a god.

Plato suggests three things: decomposition is hard (and anyone really good at it deserves adoration), any decomposition that does not lead to the discovery of things that can be recombined—composed—is counterproductive, and the separation of one thing into two should occur at "natural joints." By implication, if you decompose along natural joints—and only if you do so—you end up with objects that can be recombined into other structures. Also by implication, the natural joints occur in the domain, and "bad carving" results if you attempt to use the wrong "knife"—the wrong decomposition criterion.

If you have the right knife and are skilled in its use—know how to think about objects and about decomposition—you will complete your decomposition tasks in a manner analogous to that of the Taoist butcher:

The Taoist butcher used but a single knife, without the need to sharpen it, during his entire career of many years. When asked how he accomplished this feat, he paused, then answered, "I simply cut where the meat isn't."

---

4. Alexander, Christopher. *Notes on the Synthesis of Form*. Harvard University Press, 1970.

According to this traditional story, even meat has natural disjunctions that can be discerned by the trained eye. Of course, a Taoist butcher is like the Zen master who can slice a moving fly in half with a judicious and elegant flick of a long sword. Attaining great skill at decomposition will require training and good thinking habits. It will also require the correct knife.

Decomposition is accomplished by applying abstraction—the "knife" used to carve our domain into discrete objects. Abstraction requires selecting and focusing on a particular aspect of a complex thing. Variations in that aspect are then used as the criteria for differentiation of one thing from another. Traditional computer scientists and software engineers have used data (attributes) or functions (algorithms) to decompose complex domains into modules that could be combined to create software applications. This parallels Edsger Wybe Dijkstra's notion that "a computer program equals data structures plus algorithms."

Behind the Quotes

Edsger Wybe Dijkstra

Professor Edsger Wybe Dijkstra, a noted pioneer of the science and industry of computing, died in August 2002 at his home in the Netherlands.

Dijkstra was the 1972 recipient of the ACM Turing Award. (Some consider this award the Nobel Prize for computing.) He was a member of the Netherlands Royal Academy of Arts and Sciences and a Distinguished Fellow of the British Computer Society. He received the 1989 ACM SIGCSE Award for Outstanding Contributions to Computer Science Education. The C&C Foundation of Japan recognized Dijkstra "for his pioneering contributions to the establishment of the scientific basis for computer software through creative research in basic software theory, algorithm theory, structured programming, and semaphores." He is credited with the idea of building operating systems as explicitly synchronized sequential processes and for devising an amazingly efficient shortest-path algorithm. He designed and coded the first Algol 60 compiler.

Dijkstra is one of the best examples of the formalist position in computer science. He believed and argued in favor of the position that mathematical logic must be the basis for sensible computer program construction. He added the term *structured programming* to the language of our profession and led the fight against unconstrained "GO TO" statements in program code.

(continued)

Behind the Quotes *(continued)*

Some other common computer science concepts and vocabulary credited to Dijkstra include separation of concerns (which is important to object thinking), synchronization, deadly embrace, dining philosophers, weakest precondition, and the guarded command. He introduced the concept of semaphores as a means of coordinating multiprocessing. The Oxford English Dictionary credits him for introducing the words *vector* and *stack* into the computing context.

The fact that a computer program consists of data and functions does not mean that the nonsoftware world is so composed. Using either data or function as our abstraction knife is exactly the imposition of artificial criteria on the real world—with the predictable result of "bad carving." The use of neither data nor function as your decomposition abstraction leads to the discovery of natural joints. David Parnas pointed this out in his famous paper "On Decomposition." Parnas, like Plato, suggests that you should decompose a complex thing along naturally occurring lines, what Parnas calls "design decisions."

Both data and function are poor choices for being a decomposition tool. Parnas provided several reasons for rejecting function. Among them are the following:

- Resulting program code would be complicated, far more so than necessary or desirable.
- Complex code is difficult to understand and test.
- Resulting code would be brittle and hard to modify when requirements changed.
- Resulting modules would lack composability—they would not be reusable outside the context in which they were conceived and designed.

Parnas's predictions have consistently been demonstrated as the industry blithely ignored his advice and used functional decomposition as the primary tool in program and system design for 30 years (40 if you recognize that most object development also uses functionality as an implicit decomposition criterion). Using data as the decomposition abstraction leads to a different set of problems. Primary among these is complexity arising from the explosion in total data entities required to model a given domain and the immense costs incurred when the data model requires modification.

Note Some examples of the kind of explosion referred to in the preceding paragraph are from my own consulting practice. One organization designed a customer support system that identified 15 different customer classes because they were using a data-oriented approach and had to create new classes when one type of customer did not share attributes of the other types. In a much larger example, a company had just completed a corporate data model (costing millions of dollars) when they decided to build a very large object system. They mandated the use of the data model for identifying objects, resulting in a class library of more than 5000 classes. This became the foundation of their system, causing enormous implementation problems. A final, midrange, example was a database application for billing and invoicing wherein management demanded 1:1 replication of the existing system. This was accomplished, but it took more than a year with an offshore development team of 10 or 15 developers. My colleague and I duplicated the capabilities of the system, using object thinking, in a weekend. Management, however, was not impressed.

What criterion should be used instead of data or functions? Behavior!

Coad and Yourdon[5] claimed that people have natural modes of thought. Citing the *Encyclopedia Britannica*, they talk about three pervasive human methods of organization that guide their understanding of the phenomenological world: differentiation, classification, and composition. Taking advantage of those "natural" ways of thinking should, according to them, lead to better decomposition.

Behind the Quotes

Ed Yourdon and Peter Coad

Edward Yourdon is almost ubiquitous in the world of software development—publishing, consulting, and lecturing for decades on topics ranging from structure development to various kinds of crises (for example, the demise of the American programmer and Y2K).

(continued)

5. Yourdon, Edward, and Peter Coad. *Object Oriented Analysis*. Yourdon Press. Englewood Cliffs, NJ: Prentice Hall, 1990.

Behind the Quotes *(continued)*

The foundation for his reputation arose from his popularization of structured approaches to analysis and design. His textbook on structured analysis and design was a standard text through several editions. In 1991, he published two books, a new edition of *Structured Analysis and Design* and a small book, coauthored with Peter Coad, called *Object Oriented Analysis*.

In the object book, Yourdon made a surprising admission: the multiple model approach—data in the form of an entity relation diagram, process flow in the form of a data flow diagram, and implementation in the form of a program structure chart—advocated in his structured development writings (including the one simultaneously published) never, in his entire professional career, worked! In practice, it was impossible to reconcile the conceptual differences incorporated into each type of model.

Objects, he believed, would provide the means for integrating the multiple models of structured development into one. Unfortunately, he chose data as the "knife" to be used for object decomposition. Other ideas advanced in that book proved to be more useful for understanding objects and object thinking—especially the discussion of natural modes of thought.

Peter Coad parted ways with Yourdon after the publication of this book and developed a method and an approach to object modeling and development that was far more behavioral in its orientation. He has several books on object development that are worthy of a place in every object professional's library.

Classification is the process of finding similarities in a number of things and creating a label to represent the group. This provides a communication and thinking shortcut, avoiding the need to constantly enumerate the individual things and simply speak or think of the group. Six different tubular, yellow, and edible things become "bananas," while five globular, red, edible things become "apples." The process of classification can continue as we note that both apples and bananas have a degree of commonality that allows us to lump them into an aggregate called "fruit." In continuing the process of classification, we create a taxonomy that can eventually encompass nearly everything—the Linnaean taxonomy of living things (and its more sophisticated DNA-based successors) being one commonly known example.

Composition is simply the recognition that some complicated things consist of simpler things. Ideally, both the complicated things and the simple things they are composed of have been identified and classified. Grady Booch suggested

that all systems have a canonical form. His book *Object Oriented Design* includes a diagram captioned, "Canonical Form of Complex Systems," which captures both classification and composition hierarchies and the relationship that should exist between the two.

Classification requires differentiation, some grounds for deciding that one thing is different from another. The differentiation grounds should reflect natural ways of thought, as do classification and composition. So how *do* we differentiate things in the natural world?

Consider a tabby and a tiger. What differentiates a tiger from a tabby? Why do we have separate names for them? Because one is likely to do us harm if given the chance, and the other provides companionship (albeit somewhat fickle). Each has at least one expected behavior that differentiates it from the other. It is this behavior that causes us to make the distinction.

Some (people who still believe in data, for example) would argue that tabbies and tigers are differentiated because they have different attributes. But this is not really the case. Both have eye color, number of feet, tail length, body markings, and so on. The values of those attributes are quite different—especially length of claw and body weight—but the attribute set remains relatively constant.

Behavior is the key to finding the natural joints in the real world. This means, fortunately, that most of our work has already been done for us. Software developers simply must listen to domain experts. If the domain expert has at hand a name (noun) for something, there is a good chance that that something is a viable, naturally carved, object.

> **Note** Listening to the domain expert and jotting down nouns captures the essence of finding a natural decomposition. A full and shared domain understanding requires the negotiation of a *domain language*, a term used by Eric Evans in his forthcoming book *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

Using behavior (instead of data or function) as our decomposition criterion mandates the deferral of much of what we know about writing software and almost everything we learned to become experts in traditional (structured) analysis and design. That knowledge will be useful eventually, but at the outset it is at best a distraction from what we need to accomplish. We must relearn how to look at a domain of interest from the perspective of a denizen (user) of that domain. We need to discover what objects she sees, how she perceives them, what she expects of them, and how she expects to interact with them. Only when we are confident that our understanding of the domain and of its

decomposition into objects mirrors that of the user and the natural structure of that domain should we begin to worry about how we are going to employ that understanding to create software artifacts. (Our understanding may come one story at a time, à la XP.)

> **Note** The focus of decomposition is understanding the domain as it is. Developers and domain experts should always be aware that "what is" is not necessarily "what is best." Just because an object exists in the domain in a particular form and has specific expectations associated with it doesn't mean that the object should and must continue to exist in that form. Domains are subject to redesign, as are the objects and the relationships and communications among objects in that domain. As developers and domain experts work together, it's quite possible that they will define new objects and redesign existing objects. This is not only acceptable but highly desirable—as long as the basis for redesign activities remains the domain, not implementation environments.

Three: Objects must be composable.

As Plato noted, putting things together again is just as important as taking them apart. In fact, it is the measure of how well you took them apart. Any child with a screwdriver and a hammer can take things apart. Unless another child can look at the pieces and determine how to put them together again (or even more important, see how to take a piece from one pile and use it to replace a piece missing from another pile), the first child's decomposition was flawed.

Composability incorporates the notions of both reusability and flexibility and therefore implies that a number of requirements must be met:

- The purpose and capabilities of the object are clearly stated (from the perspective of the domain and potential users of the object), and my decision as to whether the object will suit my purposes should be based entirely on that statement.
- Language common to the domain (accounting, inventory, machine control, and so on) will be used to describe the object's capabilities.
- The capabilities of an object do not vary as a function of the context in which it is used. Objects are defined at the level of the domain. This does not eliminate the possible need for objects that are specialized to a given context; it merely restricts redefinition of that same object when

it is moved to a different context. Objects that are useful in only one context will necessarily be created but should be labeled appropriately.

- When taxonomies of objects are created, it is assumed that objects lower in the taxonomy are specialized extensions of those above them. Specialization by extension means that objects lower in the taxonomy can be substituted for those above them in the same line of descent. Specialization by constraint (overrides) might sometimes be required but almost inevitably results in a "bad" object because it is now impossible to tell whether that object is useful without looking beyond what it says it can do to an investigation of how it *does* what it says it can do.[6]

Although relatively simple to state, these requirements are difficult to satisfy. The general principle guiding the creation of composable objects is to discover and generalize the expected behavior of an object before giving any consideration to what lies behind that behavior. This is a concept that has been a truism in computer science almost from its inception. The most pragmatic consequence of this principle is the need to defer detailed design (coding) until we have a sure and complete grasp of the identification and expected behaviors of our objects (the objects relevant to the story we are currently working on) *in the domain where they live*.

Forward Thinking

A Problem of Reuse

In "Forward Thinking: Metaphor and Initial Stories," which appeared in Chapter 2, it was noted that two stories dealt with dispensing (change and product) and might involve the same objects. Further discussion of dispensing revealed three variations of a story involving some kind of dispense action: dispense a measured volume of liquid, dispense a product, and dispense change due the customer.

It would be nice if we had a single class, *Dispenser*, that could be used in all three stories. This would mean that *Dispenser* would have to be a composable object, able to be reused in different contexts without modification of its essential nature.

(continued)

6. The exception occurs when a method is declared high in the hierarchy with the explicit intent that *all* subclasses provide their own unique implementation of that method and when the details of *how* are idiosyncratic but irrelevant from the perspective of a user of that object.

Forward Thinking *(continued)*

Because of the work of the team of developers on three different stories involving dispensing, three versions of the *Dispenser* object have been created. The pairs of developers involved meet to look at each other's code and see whether they can refactor and redesign the *Dispenser* object to make it more composable—more reusable.

In one case (product dispensing), the code for the dispense method looked like the following pseudocode:

```
IF dispenserType = "Gate"
Gate open.
Else
   Set timer = 10.
   Open switch.
End-if
When timer =< 0 close switch.
```

The code in question reveals an awareness of two types of vending evident in the machines in the hall: opening a gate to drop a can of soda and pushing a product out of a coil.

The dispense method for the change dispenser looked like the following:

```
While amountToBePaid >= SmallestDenominationAvailable
AND
     AmountToBePaid > LargestDenominationAvailable
       LargestDenominationDispenser ejectCoin
        AmountToBePaid = (AmountToBePaid - largestDenomination).
```

Yes, I know you would never write code this ugly and that the second example will not really work, but code is not the issue here; refactoring is. After some discussion, the teams decided that the dispenser object was really just a façade for some mechanism that did the actual work of dispensing: a valve that opened for a period of time, a motor that ran for a period of time, or a push bar that kicked an item out of the dispenser storage area. It was also decided that the quantity to be dispensed should be supplied to the dispenser rather than calculated by the dispenser. These decisions simplified the method dramatically. In all cases, the pseudocode would look something like this:

```
For 1 to quantityToBeDispensed
     DispensingMechanism dispense.
End-loop.
```

Forward Thinking *(continued)*

The only other behaviors of *Dispenser*—to disable itself when empty or when not functioning and to identify itself—were already simple and common in all contexts. *Dispenser* was now defined in such a way as to be truly composable.

Was this accomplished only at the expense of moving some essential complexity to another object? No. Two other objects are probably involved in every dispensing operation: a collection object that contains the actual dispensers and relays dispense requests to the appropriate dispenser within the collection—a trivial behavior already built into well-designed collection objects—and a *dispensingRule* object, which is an instance of (not a subclass of) a *SelfEvaluatingRule* object. (See "Forward Thinking: Communication and Rules," for more discussion of rules in the UVM.)

Four: Distributed cooperation and communication must replace hierarchical centralized control as an organizational paradigm.

Consider one of the more widely used models in traditional software development, the program structure chart (Figure 3-1), popularized by Meillor Page-Jones.[7] At the top of the chart is the puppet master module, attended to by a court of special-purpose input, transform, and output modules. The puppet master incorporates all the knowledge about the task at hand, the capabilities of each subordinate module, and when and how to invoke their limited capabilities. The same thinking characterizes structured source code, wherein a main-line routine (frequently a *Case* statement) consolidates overall control. Each paragraph of a collection of special-purpose subroutine paragraphs is individually invoked and given limited authority to perform before control reverts to the main line.

Unlike puppet modules, objects are autonomous. They are protected from undue interference and must be communicated with, politely, before they will perform their work. It is necessary to find a different means to coordinate the work of objects, one based on intelligent cooperation among them.

It is sometimes difficult to conceive how coordination among autonomous objects can be achieved without a master controller or coordinator. One simple example is the common traffic signal. Traffic signals coordinate the movement of vehicles and people but have no awareness of what those other objects are about or even if any of them actually exist. A traffic signal knows about its own state and about the passage of time and how to alter its state as a function of elapsed time. In this model, the necessary "control" has been factored and

7. Page-Jones, Meillor. *The Practical Guide to Structured Systems Design*. Yourdon Press Computing Series. Englewood Cliffs, NJ: Prentice-Hall, Inc.1988.

distributed. The traffic signal controls itself and notifies (by broadcasting as a different color) others of the fact that it has changed state. Other objects, vehicles, notice this event and take whatever action they deem appropriate according to their own needs and self-knowledge.

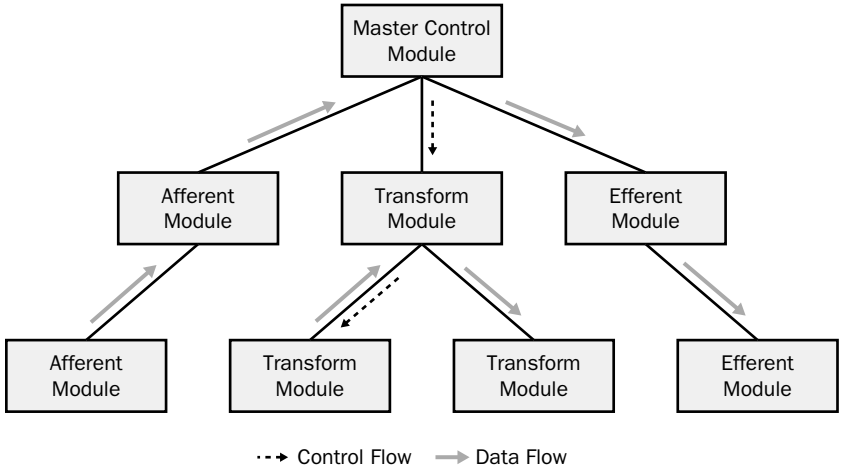


Figure 3-1 Program structure chart.

Note But what about intersections with turn arrows that appear only when needed? Who is in control then? No one. Sensors are waiting to detect the "I am here" event from vehicles. The traffic signal is waiting for the sensor to detect that event and send it a message: "Please add turn arrow state." It adds the state to its collection of states and proceeds as before. The sensor sent the message to the traffic signal only because the traffic signal had previously asked it to—registered to be notified of the "vehicle present" event. Traffic management is a purely emergent phenomenon arising from the independent and autonomous actions of a collectivity of simple objects—no controller needed. If you have a large collection of traffic signals and you want them to act in a coordinated fashion, will you need to introduce controllers? No. You might need to create additional objects capable of obtaining information that individual traffic signals can use to modify themselves (analogous to the sensor used to detect vehicles in a turn lane). You might want to use collection objects so that you can conveniently communicate with a group of signals. You might need to make a signal aware of its neighbors, expanding the individual capabilities of a traffic signal object. You will never need to introduce a "controller."

Eliminating centralized control is one of the hardest lessons to be learned by object developers.

Object Principles—Software Principles

Stating and explaining object presuppositions is important. It is also important to show the relationship between those principles and generally accepted principles of software design criteria. Exploring that relationship will further explain and illustrate the object principles and show how they recast thinking about design without rejecting traditional design goals.

Witt, Baker, and Merritt have written an excellent encapsulation of the fundamental ideas about software design and architecture.[8] Chapter 2 of their book identifies a set of generally accepted axioms and principles that define software quality:

- **Axiom of separation of concerns** Solve complex problems by solving a series of intermediate, simpler problems.
- **Axiom of comprehension** Accommodate human cognitive limitations.
- **Axiom of translation** Correctness is unaffected by movement between equivalent contexts.
- **Axiom of transformation** Correctness is unaffected by replacement with equivalent components.
- **Principle of modular design** Elaborates the axiom of separation of concerns.
- **Principle of portable designs** Elaborates the axiom of translation.
- **Principle of malleable designs** Provides the means for compositional flexibility.
- **Principle of intellectual control** Appropriate use of abstractions.
- **Principle of conceptual integrity** Suggests a limited set of conceptual forms.

Few would argue with these axioms and principles, although they would certainly argue about the appropriate means for realizing them. Object thinkers strive to achieve the goals implied by these axioms and principles as much as

8. Witt, Bernard I., F. Terry Baker, and Everett W. Merrit. *Software Architecture and Design: Principles, Models, and Methods*. Van Nostrand Reinhold, 1994.

any other software developer *and* believe that objects provide the conceptual vehicle most likely to succeed.

For example, the separation-of-concerns axiom and the principle of modularity mandate the decomposition of large problems into smaller ones, each of which can be solved by a specialist. An object is a paradigmatic specialist. Large problems (requiring a number of objects, working in concert to resolve the problem) are decomposed into smaller problems that a smaller community of objects can solve, and those into problems that an individual object can deal with. At the same time, each object addresses the principles of intellectual control (individual objects are simple and easy to understand) and the principle of conceptual integrity (there should be a small number of classes). Properly conceived, an object is a natural unit of composition as well. An object should reflect natural, preexisting decomposition ("along natural joints") of a large-scale domain into units already familiar to experts in that domain. Conceived in this fashion, an object clearly satisfies the principle of intellectual control. Objects will also satisfy the principle of conceptual integrity because there will be a limited number of classes of objects from which everything in the domain (the world) will be constructed. In Chapter 4, "Metaphor: Bridge to the Unfamiliar," an argument will be presented suggesting that the total number of objects required to build anything is around 1000.

Objects are designed so that their internal structure and implementation means are hidden—encapsulated—in order to satisfy the axiom of transformation and the principle of portable designs.

The principle of malleable designs has been the hardest one for software to realize: only a small portion of existing software is flexible and adaptable enough to satisfy this principle. In the context of object thinking, malleability is a key motivating factor. Object thinkers value designs that yield flexibility, composability, and accurate reflection of the domain, not machine efficiency; not even reusability, although reusability is little more than cross-context malleability.

In fact, it might be said that for object thinkers, all the other axioms and principles provide the means for achieving malleability and that malleability is the means whereby the highest-quality software, reflective of real needs in the problem domain, can be developed and adapted as rapidly as required by changes in the domain. Agile developers and lean developers[9] value malleability as highly as object thinkers. XP software systems emerge from software that satisfies the demands of a single story, an impossibility unless it is easy to refactor, adapt, and evolve each piece of software (each object); impossible unless each bit of software is malleable.

---

9. Poppendieck, Mary, and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers.* Addison-Wesley. 2003.

Fred Brooks wrote one of the most famous papers in software development, "No Silver Bullet: Essence and Accidents of Software Engineering."[10] In that paper, he identified a number of things that made software development difficult and separated them into two categories, *accidental* and *essential*.

Accidental difficulties arise from inadequacies in our tools and methods and are solvable by improvements in those areas. Essential difficulties are intrinsic to the nature of software and are not amenable to any easy solution. The title of Brooks's paper refers to the "silver bullet" required to slay a werewolf—making the metaphorical assertion that software is like a werewolf, difficult to deal with. Software, unlike a werewolf, cannot be "killed" (solved) by the equivalent of a silver bullet.

Brooks suggests four essential difficulties:

- **Complexity** Software is more complex, consisting of more unlike parts connected in myriads of ways, than any other system designed or engineered by human beings.
- **Conformity** Software must conform to the world rather than the other way around.
- **Changeability** A corollary of conformity: when the world changes, the software must change as well, and the world changes frequently.
- **Invisibility** We have no visualization of software, especially executing programs, that we can use as a guide for our thinking.

He also investigates potential silver bullets (high-level languages, time sharing, AI, and so on) and finds all of them wanting. Object-oriented programming is considered a silver bullet and dismissed as addressing accidental problems only.

Although I would agree with Brooks in saying that *object technology*—languages, methods, class hierarchies, and so on—addresses only accidental problems, *object thinking* does address essential difficulties, and it does so with some promise. Objects can conform to the world because their design is predicated on that world. Objects are malleable, resolving the changeability issue. Objects provide a way to deal with the complexity issue and even allow for the emergence of solutions to complex problems not amenable to formal analysis. The metaphors presented in Chapter 4 provide the tools for visualization to guide our thinking.

Object thinking suggests we deal with software complexity in a manner analogous to the ways humans already deal with real-world complexity—using behavior-based classification and modularization. Object thinking is focused on the best means for dealing with conformity and changeability issues—the

10. IEEE Computer, April 1987.

malleability principle—as a kind of prime directive. And invisibility is addressed, not with an abstract geometry as suggested by Brooks, but via simulation (working software using an XP perspective)—direct, albeit metaphorical, simulation of the real world. If we can understand the complex interactions of objects in the real world (and we do so every day), we should be able to visualize our software as an analogous interaction of objects.

Forward Thinking

Communication and Rules

Because the UVM might be dispensing food items and because we want the customer experience to be always positive, we want to ensure that no spoiled products are vended. This leads to a story—*Expire: no product is sold after its expiration date has been reached*.

The development team discusses (and codes) various ways this might be accomplished. Through a combination of refactoring efforts and arguments, it is decided that the expiration problem will best be solved by a group of objects communicating with one another, with those communications being triggered by events.

Whenever a product is placed in the vending machine, it asks itself for its expiration date. It then asks the *SystemClockCalendar* to add an *eventRegistration* (consisting of the *productID* and the "die" message) for the event generated whenever a new day is recognized by the *SystemClockCalendar*. (Programmers, even extreme programmers, often have a rather grim sense of humor; hence the "die" message to effect product expiration.) At the same time, the *Dispenser* object asks the new product to accept a registration for the "I'm dead" event that the product will generate when it receives the "die" message from its own event registration that was placed with the *SystemClockCalendar*. The dispenser's *eventRegistration* with the product will cause the message "disableYourself" to be sent to the dispenser, who will, indeed, "disable" itself (with the accompanying event that other objects—such as the menu or the dispenser collection—might register for).

Breaking up a potentially complex decision-making and cascading-effects problem into pieces that can be distributed among many objects while at the same time relying on simple, reusable components such as an *eventRegistration* and a *Dispatcher* greatly reduces the complexity that worries Brooks. It also accommodates the conformity and changeability requirements imposed on software: event registrations can be added or

Forward Thinking *(continued)*

deleted as needed, redirected to other objects, or transformed so that the registering object receives different messages at different times without the need to rewrite and recompile source code.

The development team found another opportunity for simplification as they worked with various types of rules that governed actions in different parts of the UVM. One kind of rule was, "Don't vend a product unless sufficient funds have been accumulated." Another rule was, "Refund money using the largest coins available, moving to lower-denomination coins only when the larger denomination is greater than the sum yet to be refunded."

Using a combination of refactoring and *appropriate* abstraction, the development team defined and designed a rule object. (XP philosophy warns against premature abstraction: abstraction that is not derived from refactoring, meaning not grounded in efforts to achieve simplification. Hence the adjective *appropriate* in the preceding sentence.) A rule is an ordered collection of constants, variables, and operations. A variable consists of an object and a message to be sent to that object. When a variable sends the message to the target, the resultant value replaces the unknown value of the variable. When asked to evaluate to a result, a rule iterates across its elements, asking each variable to instantiate itself to a real value, and then applies the operators to the instantiated variables and constants.

All four of Brooks's concerns about software's essential difficulties are addressed: simplification of complexity, ease of conformity and adaptability, and visualization. A rule is an easy thing to visualize—we see examples of them in everyday life frequently—and the process of instantiation and resolution is very straightforward—we can see it operating in our mind's eye with no difficulty. Reliance on simulation constantly provides other visualizations of the software we are creating.

Cooperating Cultures

Arguing for the existence of an object paradigm or object culture is not and should not be taken as an absolute rejection of traditional computer science and software engineering[11]. It would be foolhardy to suggest that nothing of value has resulted from the last fifty years of theory and practice.

11. The ideas in this section were first published in a short editorial by the author in *Communications of the ACM*, 1997.

Claiming that there are clear criteria for determining whether software is object oriented is not the same as saying all software should be object oriented. Expecting a device driver implemented with 100 lines of assembly language to reflect full object thinking is probably pointless. It's possible that Internet search engines must be implemented using "database thinking" rather than object thinking—at least for the immediate present. Different problems do require different solutions. The vast majority of problems that professional developers are paid to deliver, however, almost certainly require object solutions.

Traditional approaches to software—and the formalist philosophy behind them—are quite possibly the best approach if you are working close to the machine—that is, you are working with device drivers or embedded software. Specific modules in business applications are appropriately designed with more formalism than most. One example is the module that calculates the balance of my bank account. A neural network, on the other hand, might be more hermeneutic and objectlike, in part because precision and accuracy are not expected of that kind of system.

Traditional methods, however, do not seem to scale. Nor do they seem appropriate for many of the kinds of systems being developed 50 years after the first computer application programs were delivered. Consider the simple graph in Figure 3-2.

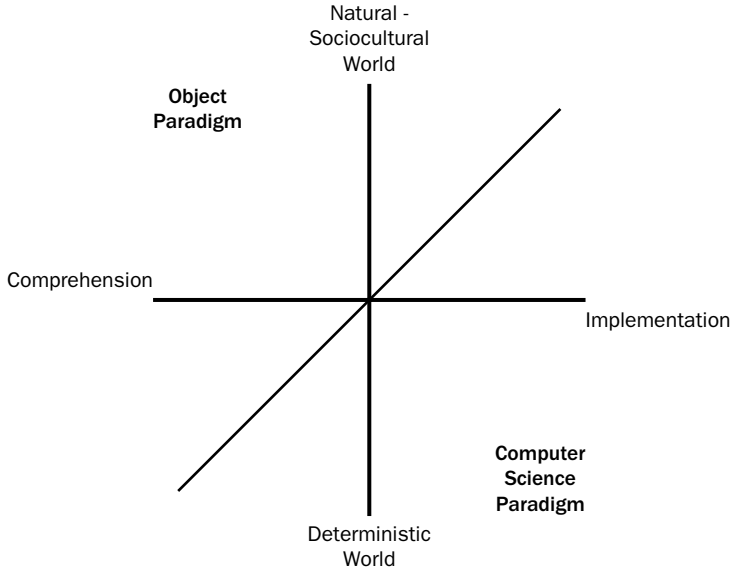


Figure 3-2 Applicability graph.

The horizontal axis represents the spectrum of activities involved in systems modeling and application development. It ranges from analysis (with the accompanying tasks of comprehending the real world, making useful abstractions, and decomposition) to implementation (compiling, testing, and executing).

The vertical axis opposes the deterministic world (the domain of hardware, discrete modules, algorithms, and small-scale formal systems) to the natural world (businesses and organizations, societies, composite systems, and cultures).

A diagonal bisects the graph to demarcate two realms. To the lower right is the realm where mainstream computer science and formalist ideas have demonstrated success. Emphasis in this realm is on defining and building hardware and using a finite set of representations (binary and operation codes) and manipulation rules (the grammar of a compiler) to implement software designs. This is the realm of formalism, where systems can be complicated and even large but not complex—that is, they do not exhibit nondeterministic behavior, they are not self-organizing, and they do not have emergent properties.

At the upper left is an area that is largely terra incognita as far as computer scientists are concerned. This is the realm of social, biological, and other complex (as that term is coming to be understood) systems. Meaning in this realm is not defined; it is negotiated. Rules are not fixed but are contextual and ephemeral. Constant flux replaces long-term consistency. This is the arena in which hermeneutic and object ideas offer an expansion of our ability to model and build systems capable of interacting with the natural systems in which they will have to exist.

This is the realm where objects and the object paradigm should dominate.

Objects provide a foundation for constructing a common vocabulary and a process of negotiated understanding of the complex world. Behavioral objects offer a decomposition technique that will yield adaptable constructs for building highly distributed, "intelligent," and flexible computer artifacts and computer-based systems.

Object thinking provides a foundation for attaining an integration of artificial systems (computer and software) with the natural systems (social and cultural contexts). Object thinking requires an awareness of the domain and the fitness of our artifacts as they operate in that domain in ways that traditional thinking ("Make the artifact meet specification") cannot. Object thinking coupled with the values of XP, especially communication, create a basis for true collaboration among users, managers, and developers.

All of this, once we learn object thinking.

This page intentionally left blank

Index

B

C

D

E

F

G

H

I

J–K

L

M

N

O

P

Q–R

S

T

U

V

W

X–Y

About the Author

Currently Dr. David West is a professor in the School of Business at New Mexico Highlands University, where he is developing an object-based curriculum in software architectures, business engineering, and management information systems. He also teaches at the University of New Mexico, where he is engaged in developing a software development track for students in graduate and undergraduate computer science.

Prior to joining the faculty at NMHU, he was an associate professor in the Graduate Programs in Software at the University of St. Thomas and a consultant/trainer to several Fortune 500 companies. He has taught courses in object-oriented development ranging from three-hour introductory sessions for managers to multiday technical seminars for professional developers, as well as semester-long courses at both the graduate and the undergraduate level.

He founded and served as the director of the Object Lab at the University of St. Thomas. The Object Lab was a cooperative effort with local corporations dedicated to researching and promoting object technology.

He was a cofounder of the Object Technology User Group (the original, not the Rational-sponsored group), the first editor of its monthly newsletter, and the principal organizer and chair of two regional conferences sponsored by OTUG.

Digitalk's Methods (the first incarnation of Smalltalk for the personal computer, later named Smalltalk/V) was his first object development environment, used to construct an "automated cultural informant," a teaching tool for cultural anthropologists learning ethnographic fieldwork techniques. His object experience is complemented by more than 20 years of software development work, ranging from assembly-language programmer to executive management.

His undergraduate education at Macalester College (oriental philosophy and East Asian history) was capped with an MS in computer science (artificial intelligence) and an MA in cultural anthropology followed by a PhD in cognitive anthropology. All the graduate degrees were earned at the University of Wisconsin at Madison.