SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 4: *Language and Software*
Sections *Wittgenstein and Software, Software Structures*

These sections examine Ludwig Wittgenstein's non-mechanistic philosophy of language and its application to software.

The entire book, each chapter separately, and also selected sections, can be viewed and downloaded at the book's website.

www.softwareandmind.com

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Published by Andsor Books, Toronto, Canada (January 2013)
www.andsorbooks.com

For disclaimers see pp. vii, xv–xvi.

Designed and typeset by the author with text management software developed by the author and with Adobe FrameMaker 6.0. Printed and bound in the United States of America.

Acknowledgements

Excerpts from the works of Karl Popper: reprinted by permission of the University of Klagenfurt/Karl Popper Library.

Excerpts from *The Origins of Totalitarian Democracy* by J. L. Talmon: published by Secker & Warburg, reprinted by permission of The Random House Group Ltd.

Excerpts from *Nineteen Eighty-Four* by George Orwell: Copyright ©1949 George Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1949 Harcourt, Inc. and renewed 1977 by Sonia Brownell Orwell, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *The Collected Essays, Journalism and Letters of George Orwell*: Copyright ©1968 Sonia Brownell Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1968 Sonia Brownell Orwell and renewed 1996 by Mark Hamilton, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *Doublespeak* by William Lutz: Copyright ©1989 William Lutz, reprinted by permission of the author in care of the Jean V. Naggar Literary Agency.

Excerpts from *Four Essays on Liberty* by Isaiah Berlin: Copyright ©1969 Isaiah Berlin, reprinted by permission of Curtis Brown Group Ltd., London, on behalf of the Estate of Isaiah Berlin.

Library and Archives Canada Cataloguing in Publication

Sorin, Andrei
Software and mind : the mechanistic myth and its consequences / Andrei Sorin.

Includes index.
ISBN 978-0-9869389-0-0

1. Computers and civilization. 2. Computer software – Social aspects.
3. Computer software – Philosophy. I. Title.

QA76.9.C66S67 2013 303.48'34 C2012-906666-4

Printed on acid-free paper.

Don't you see that the whole aim of Newspeak is to narrow the range of thought?… Has it ever occurred to you … that by the year 2050, at the very latest, not a single human being will be alive who could understand such a conversation as we are having now?

George Orwell, *Nineteen Eighty-Four*

Disclaimer

This book attacks the mechanistic myth, not persons. Myths, however, manifest themselves through the acts of persons, so it is impossible to discuss the mechanistic myth without also referring to the persons affected by it. Thus, all references to individuals, groups of individuals, corporations, institutions, or other organizations are intended solely as examples of mechanistic beliefs, ideas, claims, or practices. To repeat, they do not constitute an attack on those individuals or organizations, but on the mechanistic myth.

Except where supported with citations, the discussions in this book reflect the author's personal views, and the author does not claim or suggest that anyone else holds these views.

The arguments advanced in this book are founded, ultimately, on the principles of demarcation between science and pseudoscience developed by philosopher Karl Popper (as explained in "Popper's Principles of Demarcation" in chapter 3). In particular, the author maintains that theories which attempt to explain non-mechanistic phenomena mechanistically are pseudoscientific. Consequently, terms like "ignorance," "incompetence," "dishonesty," "fraud," "corruption," "charlatanism," and "irresponsibility," in reference to individuals, groups of individuals, corporations, institutions, or other organizations, are used in a precise, technical sense; namely, to indicate beliefs, ideas, claims, or practices that are mechanistic though applied to non-mechanistic phenomena, and hence pseudoscientific according to Popper's principles of demarcation. In other words, these derogatory terms are used solely in order to contrast our world to a hypothetical, ideal world, where the mechanistic myth and the pseudoscientific notions it engenders would not exist. The meaning of these terms, therefore, must not be confused with their informal meaning in general discourse, nor with their formal meaning in various moral, professional, or legal definitions. Moreover, the use of these terms expresses strictly the personal opinion of the author – an opinion based, as already stated, on the principles of demarcation.

This book aims to expose the corruptive effect of the mechanistic myth. This myth, especially as manifested through our software-related pursuits, is the greatest danger we are facing today. Thus, no criticism can be too strong. However, since we are all affected by it, a criticism of the myth may cast a negative light on many individuals and organizations who are practising it unwittingly. To them, the author wishes to apologize in advance.

Contents

Preface xiii

Introduction Belief and Software 1
Modern Myths 2
The Mechanistic Myth 8
The Software Myth 26
Anthropology and Software 42
Software Magic 42
Software Power 57

Chapter 1 Mechanism and Mechanistic Delusions 68
The Mechanistic Philosophy 68
Reductionism and Atomism 73
Simple Structures 92
Complex Structures 98
Abstraction and Reification 113
Scientism 127

Chapter 2 The Mind 142
Mind Mechanism 143
Models of Mind 147

Tacit Knowledge 157
Creativity 172
Replacing Minds with Software 190

Chapter 3 Pseudoscience 202
The Problem of Pseudoscience 203
Popper's Principles of Demarcation 208
The New Pseudosciences 233
The Mechanistic Roots 233
Behaviourism 235
Structuralism 242
Universal Grammar 251
Consequences 273
Academic Corruption 273
The Traditional Theories 277
The Software Theories 286

Chapter 4 Language and Software 298
The Common Fallacies 299
The Search for the Perfect Language 306
Wittgenstein and Software 328
Software Structures 347

Chapter 5 Language as Weapon 368
Mechanistic Communication 368
The Practice of Deceit 371
The Slogan "Technology" 385
Orwell's Newspeak 398

Chapter 6 Software as Weapon 408
A New Form of Domination 409
The Risks of Software Dependence 409
The Prevention of Expertise 413
The Lure of Software Expedients 421
Software Charlatanism 440
The Delusion of High Levels 440
The Delusion of Methodologies 470
The Spread of Software Mechanism 483

Chapter 7 Software Engineering 492
Introduction 492
The Fallacy of Software Engineering 494
Software Engineering as Pseudoscience 508

Structured Programming 515
The Theory 517
The Promise 529
The Contradictions 537
The First Delusion 550
The Second Delusion 552
The Third Delusion 562
The Fourth Delusion 580
The *GO TO* Delusion 600
The Legacy 625
Object-Oriented Programming 628
The Quest for Higher Levels 628
The Promise 630
The Theory 636
The Contradictions 640
The First Delusion 651
The Second Delusion 653
The Third Delusion 655
The Fourth Delusion 657
The Fifth Delusion 662
The Final Degradation 669
The Relational Database Model 676
The Promise 677
The Basic File Operations 686
The Lost Integration 701
The Theory 707
The Contradictions 721
The First Delusion 728
The Second Delusion 742
The Third Delusion 783
The Verdict 815

Chapter 8 From Mechanism to Totalitarianism 818
The End of Responsibility 818
Software Irresponsibility 818
Determinism versus Responsibility 823
Totalitarian Democracy 843
The Totalitarian Elites 843
Talmon's Model of Totalitarianism 848
Orwell's Model of Totalitarianism 858
Software Totalitarianism 866

Index 877

Preface

The book's subtitle, *The Mechanistic Myth and Its Consequences*, captures its essence. This phrase is deliberately ambiguous: if read in conjunction with the title, it can be interpreted in two ways. In one interpretation, the mechanistic myth is the universal mechanistic belief of the last three centuries, and the consequences are today's software fallacies. In the second interpretation, the mechanistic myth is specifically today's mechanistic *software* myth, and the consequences are the fallacies *it* engenders. Thus, the first interpretation says that the past delusions have caused the current software delusions; and the second one says that the current software delusions are causing further delusions. Taken together, the two interpretations say that the mechanistic myth, with its current manifestation in the software myth, is fostering a process of continuous intellectual degradation – despite the great advances it made possible. This process started three centuries ago, is increasingly corrupting us, and may well destroy us in the future. The book discusses all stages of this degradation.

The book's epigraph, about Newspeak, will become clear when we discuss the similarity of language and software (see, for example, pp. 411–413).

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from philosophy, in particular. These discussions are important, because they show that our software-related problems

are similar, ultimately, to problems that have been studied for a long time in other domains. And the fact that the software theorists are ignoring this accumulated knowledge demonstrates their incompetence. Often, the connection between the traditional issues and the software issues is immediately apparent; but sometimes its full extent can be appreciated only in the following sections or chapters. If tempted to skip these discussions, remember that our software delusions can be recognized only when investigating the software practices from this broader perspective.

Chapter 7, on software engineering, is not just for programmers. Many parts (the first three sections, and some of the subsections in each theory) discuss the software fallacies in general, and should be read by everyone. But even the more detailed discussions require no previous programming knowledge. The whole chapter, in fact, is not so much about programming as about the delusions that pervade our programming practices. So this chapter can be seen as a special introduction to software and programming; namely, comparing their true nature with the pseudoscientific notions promoted by the software elite. This study can help both programmers and laymen to understand why the incompetence that characterizes this profession is an inevitable consequence of the mechanistic software ideology.

There is some repetitiveness in the book, deliberately introduced in order to make the individual chapters, and even the individual sections, reasonably independent. Thus, while the book is intended to be read from the beginning, you can select almost any portion and still follow the discussion. An additional benefit of the repetitions is that they help to explain the more complex issues, by presenting the same ideas from different perspectives or in different contexts.

The book is divided into chapters, the chapters into sections, and some sections into subsections. These parts have titles, so I will refer to them here as *titled* parts. Since not all sections have subsections, the lowest-level titled part in a given place may be either a section or a subsection. This part is, usually, further divided into *numbered* parts. The table of contents shows the titled parts. The running heads show the current titled parts: on the right page the lowest-level part, on the left page the higher-level one (or the same as the right page if there is no higher level). Since there are more than two hundred numbered parts, it was impractical to include them in the table of contents. Also, contriving a short title for each one would have been more misleading than informative. Instead, the first sentence or two in a numbered part serve also as a hint of its subject, and hence as title.

Figures are numbered within chapters, but footnotes are numbered within the lowest-level titled parts. The reference in a footnote is shown in full only the first time it is mentioned within such a part. If mentioned more than once,

in the subsequent footnotes it is usually abbreviated. For these abbreviations, then, the full reference can be found by searching the previous footnotes no further back than the beginning of the current titled part.

The statement "italics added" in a footnote indicates that the emphasis is only in the quotation. Nothing is stated in the footnote when the italics are present in the original text.

In an Internet reference, only the site's main page is shown, even when the quoted text is from a secondary page. When undated, the quotations reflect the content of these pages in 2010 or later.

When referring to certain individuals (software theorists, for instance), the term "expert" is often used mockingly. This term, though, is also used in its normal sense, to denote the possession of true expertise. The context makes it clear which sense is meant.

The term "elite" is used to describe a body of companies, organizations, and individuals (for example, the software elite); and the plural, "elites," is used when referring to several entities, or groups of entities, within such a body. Thus, although both forms refer to the same entities, the singular is employed when it is important to stress the existence of the whole body, and the plural when it is the existence of the individual entities that must be stressed. The plural is also employed, occasionally, in its normal sense – a group of several different bodies. Again, the meaning is clear from the context.

The issues discussed in this book concern all humanity. Thus, terms like "we" and "our society" (used when discussing such topics as programming incompetence, corruption of the elites, and drift toward totalitarianism) do not refer to a particular nation, but to the whole world.

Some discussions in this book may be interpreted as professional advice on programming and software use. While the ideas advanced in these discussions derive from many years of practice and from extensive research, and represent in the author's view the best way to program and use computers, readers must remember that they assume all responsibility if deciding to follow these ideas. In particular, to apply these ideas they may need the kind of knowledge that, in our mechanistic culture, few programmers and software users possess. Therefore, the author and the publisher disclaim any liability for risks or losses, personal, financial, or other, incurred directly or indirectly in connection with, or as a consequence of, applying the ideas discussed in this book.

The pronouns "he," "his," "him," and "himself," when referring to a gender-neutral word, are used in this book in their universal, gender-neutral sense. (Example: "If an individual restricts himself to mechanistic knowledge, his performance cannot advance past the level of a novice.") This usage, then, aims solely to simplify the language. Since their antecedent is gender-neutral ("everyone," "person," "programmer," "scientist," "manager," etc.), the neutral

sense of the pronouns is established grammatically, and there is no need for awkward phrases like "he or she." Such phrases are used in this book only when the neutrality or the universality needs to be emphasized.

It is impossible, in a book discussing many new and perhaps difficult concepts, to anticipate all the problems that readers may face when studying these concepts. So the issues that require further discussion will be addressed online, at *www.softwareandmind.com*. In addition, I plan to publish there material that could not be included in the book, as well as new ideas that may emerge in the future. Finally, in order to complement the arguments about traditional programming found in the book, I plan to publish, in source form, some of the software applications I developed over the years. The website, then, must be seen as an extension to the book: any idea, claim, or explanation that must be clarified or enhanced will be discussed there.

Wittgenstein and Software

1

Ludwig Wittgenstein is regarded by many as the most influential philosopher of the twentieth century. Although he made contributions in many areas, notably in the philosophy of mathematics and the philosophy of psychology, his chief concern was language; namely, how ordinary sentences describe the world and express ideas.

Wittgenstein is famous for having created two different systems of thought, one in his youth and the other later in life, both of which greatly influenced the views of contemporary philosophers. His later ideas represent in large part a criticism and rejection of the earlier ones, and it is this change that makes Wittgenstein's philosophy so important today. For the change is, quite simply, a repudiation of the mechanistic doctrine.

His early theory – a model of language that provides an exact, one-to-one correspondence to reality – is generally considered the most rigorous system of this kind ever invented. Then, in his later philosophy, he shows not only that his earlier ideas were wrong, but also that no such system can exist. Thus, while in his early work he is attempting to find an exact linguistic representation of the world, in his later work he is trying to prove the *impossibility* of such a representation. Wittgenstein's later views, we will see presently, match the concept of complex structures and my claim that complex structures cannot be reduced to simple ones. What he is saying, in essence, is that he was wrong when he believed that complex phenomena can be represented with simple structures; that they can only be represented as systems of interacting structures; and that these systems cannot be described exactly, as can the simple structures.

Thus, unlike those philosophers who continue to believe in mechanism despite their failure to discover a useful theory, Wittgenstein created what everyone accepted as a great theory, and then saw it as his task to doubt it, and ultimately to abandon it.[1] Russell and the logical positivists, in particular, liked only his earlier theory; they rejected his later views, and persisted in the futile search for an exact linguistic representation of the world.

Wittgenstein's repudiation of mechanism has been known and studied since the 1930s, and his popularity has been increasing ever since. His ideas are

[1] Recall what we learned in "Popper's Principles of Demarcation" in chapter 3: serious thinkers *doubt* their theory and attempt to *refute* it, so they search for *falsifications*; pseudoscientists believe they must *defend* their theory, so they search for *confirmations*. Thus, Wittgenstein's shift demonstrates the value of Popper's principles.

quoted and discussed in many contexts, and have engendered an enormous body of secondary literature by interpreters and commentators. At the same time, we note that *mechanistic* theories of mind, of intelligence, of knowledge, of language, continue to flourish. Most scientists, thus, continue to represent complex human phenomena with simple structures; so they claim, in effect, that it is Wittgenstein's *early* concepts that are valid and his *later* concepts that are wrong. These scientists do not explicitly reject his non-mechanistic ideas; they simply ignore the issues he addresses in his later work, and which, if properly interpreted, clearly show the futility of searching for a mechanistic theory of mind.

In chapter 3 we saw that Popper's principles of demarcation are greatly respected, while their practical applications are largely disregarded (see pp. 230–232). The academics manage to accept and to disregard these principles at the same time by *misinterpreting* them: they ignore their value as a criterion of demarcation, and treat them instead as just another topic in the philosophy of science.

Similarly, the academics cannot reject Wittgenstein's later theory, but they cannot accept it either, because accepting it would be tantamount to admitting that their own work is merely a pursuit of mechanistic fantasies. So they resolve the dilemma by *misinterpreting* Wittgenstein's ideas: by perceiving them as a topic fit for philosophical debate, instead of recognizing their practical applications. Norman Malcolm observes that even philosophers fail to appreciate the significance of Wittgenstein's non-mechanistic ideas: "The dominant currents in today's academic philosophy have been scarcely touched [by Wittgenstein's later work, which] has been read but its message not digested. As has been aptly said, it has been assimilated without being understood."[2]

And what about our *programming* theories and practices? If they too grow out of the belief that the function of language is to map reality through one-to-one correspondence, then Wittgenstein's shift from his early to his later theory may well be the most important topic in the philosophy of software. But this shift – arguably the most celebrated event in twentieth-century philosophy, and a challenge to all mechanistic concepts of mind – is completely ignored in the world of programming. Even a casual study of our programming theories reveals that they all reflect Wittgenstein's *early* theory, which claimed that there is an exact correspondence between language and the world. They ignore the evidence he brought later to show the impossibility of such a correspondence.

[2] Norman Malcolm, *Nothing is Hidden: Wittgenstein's Criticism of his Early Thought* (Oxford: Blackwell, 1986), p. ix.

2

Wittgenstein presented his early theory in the small book *Tractatus Logico-Philosophicus*, published in 1921. Like Russell's theory of language, his theory is known as *logical atomism*; but, while the two theories are generally similar, they differ in many details. Wittgenstein started with Russell's ideas, but Russell often acknowledged that his own theory was influenced by Wittgenstein's work.

Superficially, Wittgenstein's theory makes the same assertions as the theories we have already discussed; namely, that there is a one-to-one correspondence between language and reality, and that both have a hierarchical structure. What sets his theory apart is the fact that his is the only *complete* system – simple, clear, and almost free of fallacies. Wittgenstein accomplished this feat by keeping his arguments abstract, and by excluding from his system certain types of knowledge: he insisted that it is not the task of logical analysis to search for explanations in such matters as feelings, morals, or beliefs. From all the universal language systems, Wittgenstein's is the only one that can be said to actually work. But this was achieved simply by restricting it to a small portion of reality. As he himself realized later, practically all phenomena, and the sentences representing them, must be excluded from his neat system if we want it to work.

The hierarchy that makes up Wittgenstein's system has four levels, and hence four kinds of elements: the top level is the world, which is made up of facts; facts are made up of atomic facts, and atomic facts are made up of objects. This hierarchy is perfectly mirrored in the hierarchy of language, which also has four levels and four kinds of elements: language, at the top level, is made up of propositions; propositions are made up of elementary (or atomic) propositions, and elementary propositions are made up of names. Each level and each element in one hierarchy stands in a one-to-one correspondence to the matching level and element in the other hierarchy. Thus, the totality of the world is represented by the totality of language, each fact is represented by a proposition, each atomic fact by an elementary proposition, and each object by a name. This system is illustrated in figure 4-1.[3]

Wittgenstein is careful not to define or interpret the meaning of these elements – what the objects, names, facts, and propositions actually are in the world and in language – thus leaving the system as an entirely abstract idea. He

[3] The diagram is adapted from K. T. Fann, *Wittgenstein's Conception of Philosophy* (Oxford: Blackwell, 1969), p. 20.

maintains that such definitions and interpretations are outside the scope of language, that it is impossible to convey them with precision, and that we must try to understand the system as best we can from his description. We need only recall the failure of the other language systems, which did attempt to define with precision the elements of their hierarchies, to appreciate Wittgenstein's reluctance to discuss them. We are free to interpret "names" as words and "propositions" as sentences; but we must bear in mind that, if we insist on such interpretations, the system will cease to be an abstract idea and will no longer work. Objects, names, facts, and propositions must remain, therefore, technical terms, and their use in this system must not be confused with their traditional meaning.

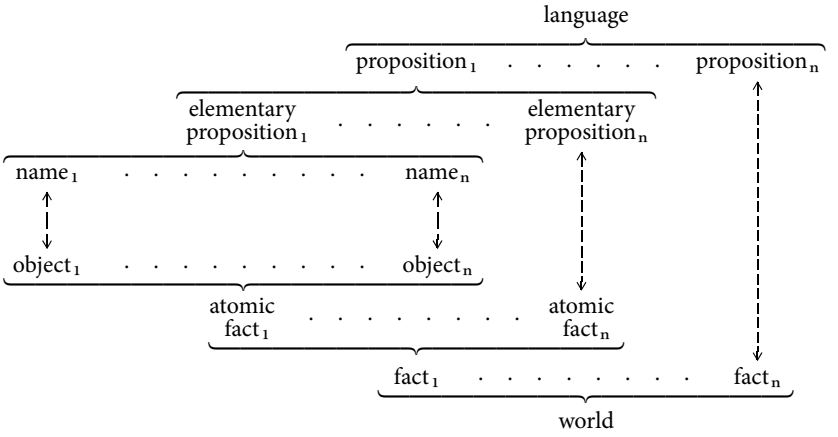

Figure 4-1

What Wittgenstein does explain is the *relations* between the various levels and elements. Objects are the most basic constituents of the world, and the names that correspond to them in language are the most basic constituents of language. In practice we may be unable to determine whether a particular entity is indeed an object or is a higher-level element, but this is unimportant. The theory simply assumes that objects exist, that the world consists ultimately of entities which are not divisible into simpler ones. And it is to these entities that the names correspond. For each object in the world there exists a name, and only one name, in language.

An atomic fact is a certain configuration of objects, and an elementary proposition is a matching configuration of names. Wittgenstein calls the relation between an atomic fact and its corresponding elementary proposition "picturing." By this he means what we called mirroring, or mapping: a one-to-

one correspondence akin to the correspondence between a map and the territory, or between a musical score and the sounds. Elementary propositions describe all possible states of affairs in the world, those that actually exist as well as those that we can only imagine. Thus, an elementary proposition can be either true or false: it is true if the atomic fact it represents exists in reality, and false if the atomic fact does not exist. An important issue, stressed by Wittgenstein, is the *independence* of elementary propositions: the truth or falsity of one elementary proposition does not depend on the truth or falsity of others. Corresponding to this, the existence or non-existence of one atomic fact is independent of other atomic facts.

At the next level, Wittgenstein's propositions are *truth functions* of elementary propositions. Truth functions – a familiar concept in logic and digital electronics – perform logical operations such as AND, OR, and NOT on one, two, or more logical operands. The operands, as well as the result of the operation, can have one of two values (called *truth values*): *False* and *True*, or a low and high state, or *0* and *1*. The operation of truth functions can be illustrated by means of *truth tables* – tables that show the result of the operation for any combination of truth values of the operands. The columns in the table denote the operands and the result, and additional columns are often included for the result of intermediate operations. The rows in the table show all possible combinations of operand values; thus, since each operand can be either *False* or *True*, there will be two rows for one operand, four for two operands, eight for three operands, and so on. The truth table in figure 4-2, for example, shows a truth function with three operands.[4]

a	**b**	**c**	**a** OR **b**	(**a** OR **b**) AND **c**
F	F	F	F	F
F	F	T	F	F
F	T	F	T	F
F	T	T	T	T
T	F	F	T	F
T	F	T	T	T
T	T	F	T	F
T	T	T	T	T

Figure 4-2

[4] The result of AND is *True* when both operands are *True*, and *False* otherwise; the result of OR is *True* when either operand is *True*, and *False* otherwise; NOT takes only one operand, and negates its value: *True* to *False*, *False* to *True*. Wittgenstein was not the first to employ truth tables, but it was his pragmatic use of them, and the popularity of his book, that established this concept in modern logic.

In Wittgenstein's system, the operands are the elementary propositions, so the truth or falsity of a proposition is completely determined by the truth or falsity of its constituent elementary propositions. In other words, depending on the truth function that defines a particular proposition, certain combinations of truth values for its elementary propositions will yield a true proposition, while the other combinations will yield a false one. (The truth table that defines a given proposition may involve a large number of operands, and therefore a huge number of rows. Remember, though, that the system is only an abstract concept.)

But propositions in language correspond to facts in the world. We already saw that the atomic facts that make up facts are in a one-to-one correspondence to the elementary propositions that make up propositions, so the truth or falsity of elementary propositions mirror the existence or non-existence of the atomic facts. Consequently, the truth function that determines the truth or falsity of a proposition also determines the existence or non-existence of the corresponding fact: whether a fact exists or not in the world depends entirely on the truth function and on the existence or non-existence of the atomic facts that make it up.

The world is as it is, and the function of language is to describe it. All possible facts, and all possible propositions, no matter how complex, can be expressed as truth functions of atomic facts and elementary propositions, respectively. This system, therefore, represents both the world and the language that mirrors it. It represents even facts that do *not* exist in the world: these are the combinations of atomic facts for which the truth functions yield *False*. Facts that do not exist are mirrored in language by combinations of elementary propositions for which the same truth functions yield *False*: false assertions ("snow is black," "the sun moves round the earth"), fictional stories, and the like.

All these propositions, says Wittgenstein, true or false, are *meaningful* propositions. They must be distinguished from those propositions that cannot be expressed as truth functions (because their truth or falsity does not depend exclusively on the truth or falsity of some elementary propositions). For example, no truth function can describe the situation where the same combination of truth values for the elementary propositions yields, unpredictably, sometimes *True* and sometimes *False*. And they must also be distinguished from those propositions that simply say nothing; for example, a proposition that is always *True*, regardless of the truth values of its elementary propositions (a tautology), or one that is always *False* (a contradiction). Such propositions, says Wittgenstein, are *meaningless*, because they do not mirror facts that either exist or do not exist in the world.

Meaningless propositions, it turns out, form a large part of our discourse:

philosophy, religion, ethics, metaphysics, and much of everyday language consist chiefly of such meaningless propositions. Even logic and mathematics consist of meaningless propositions, because they do not represent facts, but are self-contained deductive systems: they are purposely designed so that, if one follows their rules, one always expresses the truth. Thus, purely deductive systems assert nothing about the world. Only the propositions of empirical science can be said to be meaningful, to mirror facts. Wittgenstein does not deny the value of the other propositions; he merely says that they do not reflect facts from the real world. To say that they are meaningless overstates the case, but this uncompromising position – this arrogance – was an important aspect of his early philosophy.

❖

Although only an abstract concept (unlike the language systems we examined earlier), Wittgenstein's theory was received with enthusiasm. Everyone was fascinated by its simplicity and by its apparent power to explain the world, and it was generally seen as a real improvement over the others.

But Wittgenstein's system is no improvement. Like the others, it is an attempt to determine by strictly mechanical means, through a logical analysis of linguistic elements and without taking into account the context in which they are used, whether or not the facts they represent exist in the world. It seems to be an improvement because it restricts the universe of meaningful propositions to a small fraction of those used in ordinary discourse. Thus, his neat system appears to work because Wittgenstein permits into it only those aspects of the world that *are* neat, while branding everything else as meaningless.

In the end, very little is left that is *not* considered meaningless. In fact, even the propositions of empirical science must be excluded, because they can rarely be reduced to the system's terminal elements – to names pointing to simple, irreducible objects. (In other words, the system is incompatible with the fundamental tenet of empirical science – the requirement that propositions be accepted only if they can be verified through direct observation.) Thus, similarly to the ideas of logical positivism (see p. 324), Wittgenstein's view of meaningfulness is in effect a criterion of demarcation based on verifiability. Popper, who held that a criterion of demarcation must be based on falsifiability (see "Popper's Principles of Demarcation" in chapter 3), pointed out this weakness shortly after the *Tractatus* was published.[5]

[5] Karl R. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge*, 5th ed. (London: Routledge, 1989), pp. 39–40.

❖

It should be obvious why Wittgenstein's early theory can help us to understand the origin of our software delusions. If we accept his system as the best expression of the mechanistic language delusion – the belief that language mirrors the world through one-to-one correspondence, and that both can be represented with hierarchical structures – we recognize our software delusions as an embodiment of Wittgenstein's theory.

The programming theories claim that the world can be mirrored perfectly in software if we design our applications as hierarchical structures of software elements: operations, blocks of operations, larger blocks, modules. To use Wittgensteinian terminology, these elements are software propositions that correspond to specific facts in the world, each proposition consisting of a combination of lower-level propositions. In a perfect application, the software propositions at each level are independent of one another; and the truth or falsity of each one (that is, whether or not it mirrors *actual* facts) is completely determined by its lower-level, constituent propositions. Each software element is, in effect, a truth function of its constituent elements. (Most software applications, however, need more than the two levels that Wittgenstein allows for propositions in his system.)

These theories fail because they try to represent with neat structures, facts that are not independent and are not made up of elementary facts in the precise way expected by the software propositions. Just as Wittgenstein's theory accepts only a small fraction of the totality of propositions, and brands the rest as meaningless, the programming theories accept only a small fraction of the totality of *software* propositions: those corresponding to facts that *can* be represented with neat structures of lower-level facts. Most situations in the real world, however, are not neatly structured. Most situations can only be mirrored with software propositions that are, in the Wittgensteinian sense, meaningless: propositions that *cannot* be expressed as exact functions of independent lower-level propositions.

Unfortunately, we cannot write off these situations as Wittgenstein does in his system. For, if we did, there would be practically no situations left for which software applications are possible. Thus, what is wrong with the programming theories is not that they let us create *bad* applications, but on the contrary, that they restrict us to *logically perfect* applications: to simple hierarchical structures of software propositions. And, just as is the case with the meaningful propositions in Wittgenstein's system, we can represent with logically perfect applications only a small fraction of the facts that make up the world.

3

Let us examine next how Wittgenstein changed his views, and what his new philosophy of language means to us. After writing the *Tractatus*, Wittgenstein felt that he had no further contribution to make to philosophy, and for nearly ten years he pursued other interests. Although his theory was becoming increasingly popular and influential, he himself was becoming increasingly dissatisfied with it. By the time he returned to philosophy he had new ideas, and he continued to develop them for the rest of his life. He wrote only one book expounding his later philosophy: the posthumously published *Philosophical Investigations*. Several other books, though, consisting largely of transcripts of his notes and lectures, have been published since then. As in the *Tractatus*, his ideas are interconnected and cover many fields, but we are interested here only in his central concern: how language represents knowledge, thought, and reality.

Wittgenstein admits now that, if we want to understand how language mirrors reality, we cannot restrict ourselves to neat linguistic structures and formal logic. We cannot assume that only those propositions which can be expressed with truth functions are meaningful. All normal uses of language have a meaning, simply because they fulfil certain social functions. So, instead of *ignoring* those propositions that cannot be reduced to neat structures of linguistic entities, we should conclude that they cannot be so reduced because the reality they mirror cannot be reduced to neat structures of facts: "The more narrowly we examine actual language, the sharper becomes the conflict between it and our requirement [i.e., our wish]. (For the crystalline purity of logic was, of course, not a *result of investigation*: it was a requirement [i.e., a wish].)"[6]

Wittgenstein notes that complex entities are complex in more than one way; that is, there are several ways to break down a complex entity into simpler parts. It is impossible, therefore, to define a precise, unique relationship between the parts and the complex whole.[7] If the complex entities are facts, or propositions that mirror these facts, there is always more than one way to represent these facts and propositions as a function of simpler facts and propositions. So if we want to depict these entities with a hierarchical structure, we will find *several* hierarchies through which a particular complex entity is related to simpler ones.

[6] Ludwig Wittgenstein, *Philosophical Investigations*, 3rd ed. (Englewood Cliffs, NJ: Prentice Hall, 1973), §107. [7] Ibid., §47.

A chessboard, for example, is a complex entity: we can view it as a configuration of sixty-four squares, but we can also view it as eight rows of squares, or as eight columns, or as various arrangements of pairs of squares, or as combinations of squares and colours.[8] Clearly, there are many ways to describe a chessboard as a structure of simpler elements, and all these structures exist at the same time.

A more difficult example is the notion of a game.[9] Each of the various activities we call games has a number of distinguishing characteristics, but there is no one characteristic that is common to all of them. There are ball games, board games, card games, and so on; some are competitive, but others are mere amusements; some call for several players, while others are solitary; some require skill, and others luck. This means that there is no set of principles that would permit us to determine, simply by following some identification rules, whether a given activity is or is not a game. But, in fact, even without such principles, we have no difficulty identifying certain activities as games. So it seems that games have a number of similarities, for otherwise we would be unable to distinguish them as a specific type of activity. And yet, despite these similarities, we cannot represent them through a neat classification of activities.

Just like the word "game," says Wittgenstein, the meaning of most words and sentences is imprecise; it depends largely on the context in which we use them. The meaning of linguistic entities is imprecise because the reality they mirror is an imprecise structure of facts. No methods or rules can be found to relate all the meanings of words and sentences, in all conceivable contexts; so we must think of them simply as *families* of meanings. Thus, Wittgenstein coined the phrase "family resemblance" to describe the complex relationship between facts, or between the linguistic entities that correspond to facts: "And the result of this examination is: we see a complicated network of similarities overlapping and criss-crossing.... I can think of no better expression to characterize these similarities than 'family resemblances'; for the various resemblances between members of a family: build, features, colour of eyes, gait, temperament, etc. etc. overlap and criss-cross in the same way."[10]

No methods or rules can describe *all* the resemblances between the members of a family. Their family relationship can be accurately represented, of course, with the hierarchical structure known as the family tree. But this relationship reflects only *one* of their attributes. If we classified in a hierarchical structure the relationship that reflects another attribute (height, or eye colour, or a particular facial feature), that hierarchy will not necessarily match the family tree. Each attribute gives rise to a different classification. Thus, the

[8] Ibid. [9] Ibid., §66. [10] Ibid., §§66–67.

resemblance of family members is the result of several hierarchies that exist at the same time – hierarchies that overlap and intersect.

Similarly, if we classified the activities we call games on the basis of one particular attribute (number of players, level of skill, use of a ball, etc.), we would end up with a different hierarchy for each attribute. It is impossible to create a hierarchy with *game* as the top element, the individual games as terminal elements, and their categories as intermediate elements – not if we want to capture in this hierarchy *all* their attributes. Instead, there are *many* such hierarchies, a different one perhaps for each attribute. All have the same top element and the same terminal elements, but different intermediate elements. And all exist at the same time.

This, incidentally, is true of all classifications. We classify plants and animals, for example, into a hierarchy of classes, orders, families, genera, and species on the basis of *some* of their attributes, while ignoring the others. It is impossible to capture *all* their attributes in one hierarchy. The current hierarchy is useful to biologists, but we could easily create different ones, on the basis of other attributes. Since the plants and animals are the same, all these hierarchies exist at the same time.

We must take a moment here to clarify this point. When I say that one hierarchy cannot capture all the attributes of entities like games, what I mean is that a *correct* hierarchy cannot do so. For, one can always draw a tree diagram in the following manner: assuming for simplicity only three attributes and only two or three values for each attribute, we divide games into, say, ball games, board games, and card games; then, we divide *each one* of these elements into games of skill and games of luck, thus creating six elements at the next lower level; finally, we divide *each one* of these six elements into competitive and amusing, for a total of twelve categories. This structure is shown in figure 4-3.

Clearly, if we limit games to these three attributes, any game will be a terminal element in this structure, since it must fit within one of the twelve categories: ball games that require skill and are competitive, ball games that require skill and are amusing, etc. The intermediate elements are the categories that make up the various levels; and the attributes are represented by the branches that connect one category to the lower-level ones.[11]

[11] Note that it is irrelevant for the present argument whether we treat the use of a ball, a board, and cards as three values of one attribute (on the assumption that no game requires more than one of them), or as three separate attributes (each one having two values, *yes* and *no*); only the intermediate levels and elements of the classification would differ. This is true in general, and I will not repeat it in the following discussion. We can always reduce a classification to attributes that have only two values (*use* or not, *possess* or not, *affect* or not, etc.), simply by adding levels and categories. Thus, since logically there is no difference between multivalued and two-valued attributes, I will use both types in examples.

This structure, however, although a tree diagram, is not a true hierarchy. It captures all three attributes, but it does so by *repeating* some of them; *competitive*, for instance, must appear in six places in the diagram. While such a classification may have its uses, it does not reflect reality: it does not represent correctly the actual categories and attributes. We don't hold in the mind, for example, six different notions of competitiveness – one for ball games that require skill, another for ball games that require luck, and so forth. We always know in the same way whether a game is competitive or amusing; we don't have to analyze its other attributes first. To put this in general terms, we don't perceive the attributes of games as *one within another*.

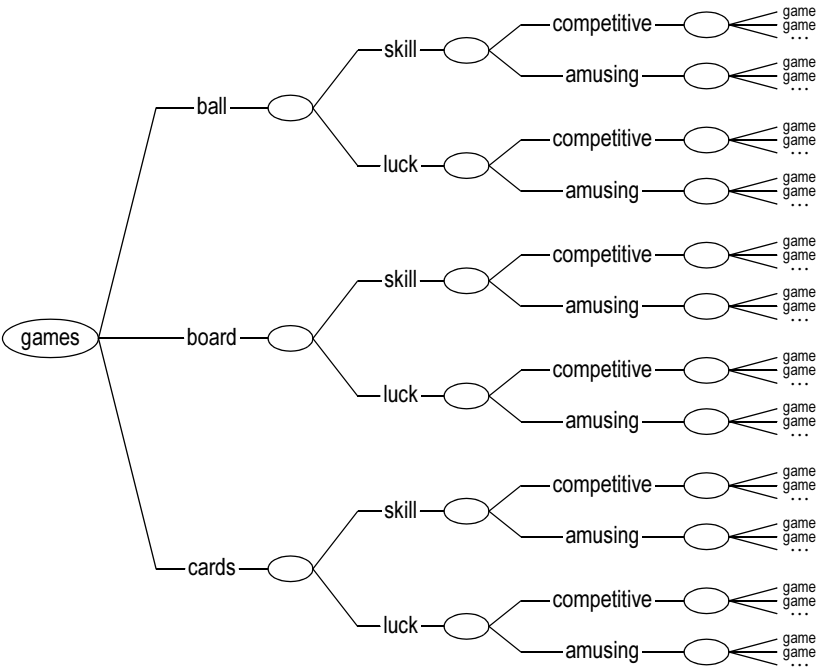

Figure 4-3

There is another way to look at this problem. The order in which we showed the three attributes was arbitrary. We can create another tree diagram by dividing games first into competitive and amusing, then each one of these two elements into games of skill and games of luck, and then the resulting four elements into ball games, board games, and card games, as in figure 4-4; and we end up with the same twelve elements as before. If *this* structure reflected reality, rather than the first one, we would indeed have only one way to perceive

competitive and amusing games; but now we would be using four different methods to decide whether a game is a card game: one for competitive games that require skill, another for competitive games that require luck, and so forth. Again, this is silly: we always know in the same way whether a game is a card game. And, even if we were willing to admit that this is how we distinguish games, why should we prefer one way of arranging the attributes rather than the other? Since this situation is absurd, we must conclude that this is *not* how we perceive games, so this type of structure does not reflect correctly our knowledge of games and their attributes. A correct hierarchy must include each attribute only once, and no such hierarchy exists.

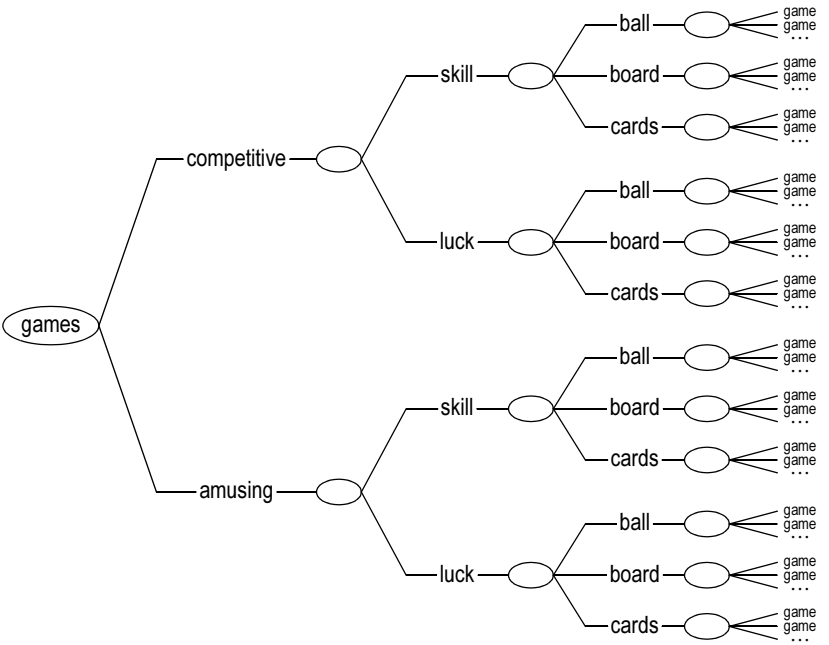

Figure 4-4

Thus, it seems that we can distinguish games *in our mind* on the basis of several attributes simultaneously, but we cannot represent this phenomenon with a simple hierarchical structure. The only way to represent it is as *several* structures, one for each attribute, while remembering that our mind does not, in fact, perceive these structures separately (see figure 4-5). The top element (the concept of games) and the terminal elements (the individual games) are the same in all three structures, and they are also the same as in the previous structures.

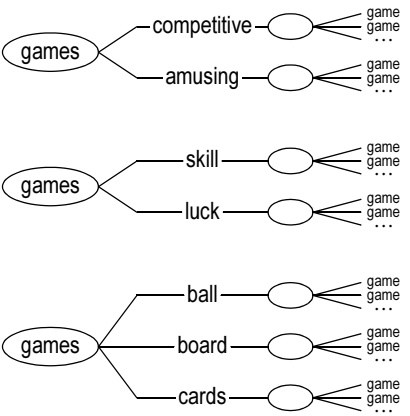

Figure 4-5

Intuitively, we can easily see why it is impossible to depict all the attributes in one hierarchy. If the attributes are independent – if the possession of a certain attribute by a game is independent of its possession of other attributes – then any attempt to depict all the attributes in one hierarchy is bound to distort reality, because it must show some attributes as subordinate to others.

I used the concept of hierarchical classifications in the foregoing analysis because it is especially suitable for studying Wittgenstein's problem of family resemblance. This concept was introduced in chapter 1, along with the concept of complex structures (see pp. 100–104). But I want to stress that this analysis is new – it is *not* how Wittgenstein, or the many philosophers who interpreted his work, studied the problem. They did little more than *describe* it. The problem, again, is to understand how our mind discovers that several entities are related through their attributes even though no one attribute has the same value for all entities, and to understand also why it is impossible to represent this apparently simple phenomenon mechanistically. The concept of hierarchical classifications can be seen, therefore, as a model for studying Wittgenstein's problem. And, while still informal, this model provides a more accurate depiction than the discussions found in philosophy texts. (We will make good use of this concept later, for both language and software structures.)

❖

So, Wittgenstein concludes, we must give up the idea of reducing facts, and the linguistic entities that mirror these facts, to perfect hierarchical structures: "We see that what we call 'sentence' and 'language' has not the formal unity that I

imagined, but is the family of structures more or less related to one another."[12] Wittgenstein coined the phrase "language games" to describe the relationships that turn linguistic entities – sentences, expressions, even individual words – into families of entities. Language games replace in his new theory what was in his earlier system the neat hierarchy of propositions, and serve to mirror in language the "games" that exist in the world: the complex structures of facts that replace in the new theory the neat hierarchy of facts of the earlier system.

All activities performed by people in a society are, in the final analysis, akin to games: they form structures that are indefinite and overlapping, rather than formal and independent. The linguistic structures are an inseparable part of these activities, so they too are indefinite and overlapping. Thus, we cannot state with precision what is a meaningful proposition, simply because we cannot find any characteristics that are common to all meaningful propositions. The concept of language can only be defined informally, as a collection of interrelated linguistic structures: "Instead of producing something common to all that we call language, I am saying that these phenomena have no one thing in common which makes us use the same word for all – but that they are *related* to one another in many different ways. And it is because of this relationship, or these relationships, that we call them all 'language.'"[13]

Like the propositions of the earlier system, the totality of possible language games forms the structure we recognize as language. Also like the propositions, language games do it by creating complex elements from simpler ones, and thus higher levels of abstraction from lower ones. But this is no longer a simple hierarchical structure. The relations that generate the elements at one level from those at the lower level cannot be defined with precision, as could the truth functions of the earlier system. Because the elements are related through several structures simultaneously, the only way to describe this relationship is informally, as "family resemblance."

Wittgenstein's "family resemblance" is, obviously, the type of phenomenon that requires the model of a complex structure. Recall our discussion of complex structures in chapter 1 (pp. 98–104). We saw that any entity has a number of attributes, each one relating it to other entities which have that attribute. Any entity, therefore, is an element in a different structure for each one of its attributes. But all these structures exist at the same time; so they interact, because they share these elements.

The family tree is a perfect hierarchy, but this structure alone does not determine all the resemblances. We could classify the members of a family according to other attributes, and each classification would be a different structure. These structures exist at the same time and share their terminal

[12] Wittgenstein, *Philosophical Investigations*, §108. [13] Ibid., §65.

elements (the people themselves), so they interact. The phenomenon of family resemblance is a complex structure, and cannot be described precisely and completely as a function of the people.

We note the same situation in the relations between games – the phenomenon that inspired the phrase "language games." A game is related to other games through all its attributes, and each attribute gives rise to a different structure. The games are the terminal elements shared by these structures, and this phenomenon – the existence of activities perceived as games – is a complex structure. It is impossible to describe the concept of games, precisely and completely, as a function of the individual games.

The incorrect hierarchies in figures 4-3 and 4-4 reflect, then, the futility of attempting to reduce a complex structure to simple ones. Also, since simple structures are logically equivalent to deterministic systems, the impossibility of depicting several attributes in one structure demonstrates the indeterministic nature of these phenomena.

❖

What Wittgenstein's new theory does, essentially, is mirror in language a greater portion of reality than did his earlier theory. Propositions mirror facts, and Wittgenstein acknowledges now that there exist facts which cannot be expressed with precision as a function of simpler facts. The only way to mirror them in language is by *permitting* those propositions which the earlier theory branded as meaningless; that is, those propositions which *cannot* be expressed as a logical function of simpler linguistic elements. Such propositions are not meaningless, Wittgenstein says now. Language is a social device, and its function is to assist us in our everyday activities. A proposition must be deemed meaningful, therefore, simply if it helps us to perform an act, or if it describes an aspect of human life. The imprecise language we use in everyday discourse is a reflection of the imprecise nature of our life, so we must accept it as meaningful. The meaning of a word or sentence must be determined by its use, not through formal logic.

The price we pay for mirroring in language a greater portion of reality is having to give up the preciseness of the earlier theory: we must be content with the imprecise concept of language games. To use our own terminology, the complexity of the world cannot be represented accurately enough with mechanistic models, so we need non-mechanistic ones, which must remain informal.

It is significant that Wittgenstein did not attempt to improve his earlier theory by *expanding* it: he did not attempt to make the theory match more facts by adding rules, or levels, or types of elements. When he realized

that a mechanistic theory cannot work, he did not hesitate to abandon the mechanistic dogma. This attitude stands in sharp contrast to the attitude of the other thinkers who start with mechanistic theories. Scientists who search for mechanistic theories of language, mind, and society, or software experts who invent mechanistic programming theories, continue to defend mechanism even when they see their theories falsified. They resort then to the pseudoscientific practice of *expanding* their theories: they add more and more features to make the theories cope with those conditions that would otherwise falsify them (see pp. 225–226).

Wittgenstein recognized the dishonesty and futility of these attempts, and this is why his work is so important today. He started with the same mechanistic delusions and created a great mechanistic theory; but unlike the other thinkers, he realized that the methods of the exact sciences cannot explain language and the mind, and settled for a less ambitious, and less formal, theory – a non-mechanistic one. If "explanation" means exact methods and theories, he frequently asserts, then the phenomenon of language cannot be *explained*, but only *described*: "We must do away with all *explanation*, and description alone must take its place."[14]

S. S. Hilmy,[15] after studying Wittgenstein's unpublished manuscripts and notes, concludes that his later philosophy is essentially a rejection of the prevailing notion that the "scientific way of thinking" – namely, reductionism and atomism – is as important in the study of human minds as it is in physics. Wittgenstein's philosophy, thus, is a struggle against the scientistic current of his time – a current in which he himself had been caught earlier, and which, despite his legacy, the passage of more than half a century, and the failure of countless mechanistic theories of mind, is just as strong today.

4

What can we learn from Wittgenstein's later philosophy that can help us in our software pursuits? We already saw that mechanistic programming theories are in the domain of software what Wittgenstein's *early* theory is in the domain of language. The early theory claims that language mirrors reality, that both can be represented with perfect hierarchical structures, and that there is a one-to-one correspondence between propositions and facts – between the levels and elements that make up language and those that make up the world. Similarly,

[14] Ibid., §109.

[15] S. Stephen Hilmy, *The Later Wittgenstein: The Emergence of a New Philosophical Method* (Oxford: Blackwell, 1987), esp. ch. 6.

the programming theories claim that software applications mirror reality through neat hierarchical structures of software entities (operations, blocks of operations, modules), which correspond on a one-to-one basis to the facts that make up our affairs.

When the programming theories insist that applications be designed as neat structures of entities within entities, they do so because of our belief that the world can be represented with neat structures of things within things – the same belief that engenders mechanistic *language* theories. Since it is the same world that we want to mirror in language and in software, it is not surprising that we end up with similar language and software theories.

In his later theory, Wittgenstein shows us that the world cannot be represented with formal hierarchical structures; that facts are related to other facts in many different ways at the same time; and that these complex relationships can only be expressed informally – as families, or systems, of facts. Then he shows us that, since language mirrors the world, we find similar relationships in language. This is why everyday discourse consists of informal language games and families of linguistic entities, rather than formal structures of propositions.

But if it is the same world that we try to mirror in our software applications, why do we expect simple *software* structures to succeed where simple *linguistic* structures fail? Our programming theories can be no more formal, no more exact, than our language theories. Following Wittgenstein, we could call our software applications – the structures of operations and modules – software games; and we must accept the fact that these applications cannot be the neat hierarchical structures we wish them to be.

The fundamental principle of software mechanism is that the entities which make up an application constitute *only one* logical structure. We are told that applications must be designed as perfect hierarchical structures, so we draw block diagrams and flowcharts that depict operations within operations, modules within modules. Having done this, we are convinced that the software modules themselves are as independent from one another as the blocks which represent them on paper. We are convinced, in other words, that the modules are related to one another only through the lines connecting the blocks in the diagram. We conclude, then, that the main difficulty in programming an application is the process of analysis: the discovery of the particular hierarchical structure which corresponds to the structure of facts that make up the requirements. For, once we establish this one-to-one correspondence between software and reality – once we discover the software entity that corresponds to each entity in our affairs, from the most general to the simplest – it is relatively easy to translate the resulting structure into a programming language.

This principle, stated in one form or another, forms the foundation of all

programming theories. But the principle is invalid, because it is impossible to reduce reality to a hierarchical structures of facts, and perforce impossible to reduce software applications to a hierarchical structures of software entities. The principle is invalid because facts form, not one, but many hierarchical structures. Software entities, therefore, if they are to reflect the facts, must also be related through many structures at the same time.

We saw that real entities (objects, processes, events, concepts – facts, in Wittgensteinian terminology) have a large number of attributes, and are elements in a different structure through each attribute. The corresponding software entities, too, have a large number of attributes (using files and variables, calling subroutines, being affected by business rules, etc.), and are elements in a different structure through each attribute. Thus, the block diagram we believe to depict the application's logic is merely *one* of these structures. No matter how strictly we design the application as independent entities (as a neat structure of operations within operations, modules within modules), these entities will also be related through other structures, besides the structure we see in the diagram. All these structures exist at the same time and share their elements – those software entities thought to be independent. An application, therefore, is a complex structure, just like the reality it mirrors, and no diagram can capture *all* the relations between the software entities.

That structure we perceive to be the application's logic – let us call it the *main* structure – is perhaps the most obvious, and may well represent some of the most important operations or relations. But just because the other structures are less evident, and we choose to ignore them, it doesn't mean that they do not affect the application's performance.

Note how similar this phenomenon is to the phenomenon of language. Nothing stops us from inventing theories based on *one* of the structures that make up sentences: their syntactic structure, like Chomsky, or their logical structure, like Russell and Carnap. But this will not provide an explanation of language. These theories fail because language consists of many structures, not one, and it is their totality that determines the meaning of sentences. Moreover, it is not only linguistic structures that play a part in this phenomenon. The structures formed by the other knowledge present in the mind, and those formed by the context in which the sentences are used, are also important. And the same is true of software: the performance of an application is determined, not only by the syntactic structure formed by its elements, or by the logical structure that is the sequence of their execution, but by *all* the structures through which they are related.

Wittgenstein discusses many situations where reality cannot be reduced to one simple structure, but his two famous examples – games and families, which we examined previously – already demonstrate the similarity of language and

software. Thus, there is no one hierarchical structure with the concept of games as the top element and the individual games as the terminal elements – not if we want to classify games according to *all* their attributes. Instead, we find *many* structures with these top and terminal elements. Nothing stops us from considering one of them, perhaps the one dividing games into competitive and amusing, as the only important structure; but this will not explain completely the concept of games.

Similarly, there is no one hierarchical structure with a particular software application as the top element and some elementary operations as the terminal elements – not if we want to describe the application completely. There are *many* such structures, each one reflecting a different aspect of the application. Nothing stops us from interpreting the application's block diagram or flowchart as its definition, or logic; but if we do, we should not be surprised if it does not explain its performance completely. (The structures that make up software applications are the subject of the next section.)

Software Structures

1

To understand how the software entities that make up an application can form several hierarchical structures at the same time, think of them as similar to *linguistic* entities, since both are reflections of entities that exist in the world. A software entity can be a module, a block of statements, and even one statement. And, insofar as they reflect real processes or events, the software entities possess, just like the real entities, not one but *several* attributes; so they must belong to a different structure through each one of these attributes. The attributes of a software entity are such things as the files, variables, and subroutines it uses. Anything that can affect more than one software entity is an attribute, because it relates these entities logically, thereby creating a structure.

I want to discuss now some common principles we employ in our applications, and which I will call simply *software principles*. What I want to show is that it is these principles that endow software entities with attributes, and serve, therefore, to relate them.[1]

A software principle, then, is a method, a technique, or a procedure used in the art of programming. An example is the sharing of data by several elements of the application; this principle is implemented by means of database fields

[1] As is the case throughout this book, the terms "entity" and "element" refer to the same things, and are usually interchangeable. See p. 99, note 1.

and memory variables. Another example is the sharing of operations; this principle is typically implemented by means of subroutines. A principle frequently misunderstood is the sequence in which the application's elements are executed by the computer; this principle is implemented through features found, implicitly or explicitly, in each element. The user interface, and the retrieval of data through reports and queries, are also examples of principles. Lastly, the methods we use to represent our affairs in software constitute, in effect, principles. The principles we will study in this section are: practices, databases, and subroutines. (We will further study software principles later in the book, particularly in chapter 7, when discussing various software theories.)

I will refer to the individual instances of software principles as *software processes*: each case of shared data or shared operations, each business practice, each report or query or user interface, is a process. And each process endows the application's elements with a unique attribute, thereby creating a unique structure – a unique way to relate these elements. Ultimately, these structures reflect the various *aspects* of the application: each aspect corresponds to a process, and each process gives rise to an attribute, and hence a structure. An application may comprise thousands of such structures.

2

Let us start with those processes that reflect the various *practices* implemented in the application – the countless rules, methods, and precepts that are embodied in the application's logic. Practices can be divided into two broad categories: those related to the activities we want to translate into software, and those related to the methods we employ in this translation. Let us call the first category *business practices*, and the second one *software practices*. Business practices include such processes as the way we invoice a customer, the way we deal with surplus inventory, and the way we calculate vacation pay. And software practices include such processes as reporting, inquiry, file maintenance, and data entry.

Practices necessarily affect software elements from different parts of the application. They create, therefore, various relations between these elements – relations that do not parallel the relations created by the main structure. If, for example, we depict with a hierarchical diagram one of the practices implemented in the application, the diagram will not be a distinct section of the main diagram, although the practice *is* part of the application's logic. The implementation of practices, thus, creates *additional* structures in the application – structures that share their elements with the main structure.

As an example of practices, consider the case of back orders. One way a

distributor can handle back orders is as follows: when the quantity ordered by a customer exceeds the quantity on hand, the order is placed on hold, the customer is informed, special messages and forms are generated, and so on; the order is revived later, when the necessary quantity becomes available. Another way to handle back orders, however, is by allowing the quantity on hand to go negative: the order is processed normally, thus reducing the quantity on hand below zero; steps are taken to ensure the product in question is ordered from the supplier before the order's due date; when the product is received, the quantity on hand is restored to zero or to a positive value.

It is obvious that the implementation of a back-order process will not only affect *many* elements of the application, but will affect them in different ways depending on the back-order method chosen. And it is just as obvious that the relations between these elements, as seen from the perspective of the back-order process alone, may not be the same as the relations created by the application's main structure. The main structure will probably reflect such aspects of reality as the functions selected by users from a menu, the responsibilities of the different departments, or the daily operating procedures. Thus, while the software elements that make up the application must reflect the main structure, some of these elements must also reflect the particular back-order process chosen: variables and fields may or may not have negative values, back-order data is or is not printed, purchase orders are or are not issued daily, and so on.

This sharing of elements, moreover, will not be restricted to the high levels of the structure, but will affect elements at all levels, from entire modules down to individual operations. The back-order process, in other words, cannot be implemented as an independent piece of software connected to the rest of the application simply through some input and output links. If we represent the application with a block diagram that depicts the main structure, we will not find the back-order process as one particular block in the diagram; rather, it is part of many blocks, and it connects therefore these blocks through a structure that is different from the one depicted by the diagram.

To understand why a process is in fact a structure, think of the application from the perspective of this process alone, while ignoring all its other functions. Seen from this perspective, the application can be depicted as a hierarchical structure that divides the application's elements into two categories, those affected and those unaffected by the process. The former may then be classified into further categories, on lower and lower levels, according to the way the process affects them, while the latter may be classified into categories reflecting the reasons why they are unaffected. Finally, the lowest-level elements will be the individual statements that make up the application. But these elements are also the elements used by the main structure, so the

application can be said to consist of either structure, or of both structures at the same time.

The back-order process is only an example, of course, only one of the hundreds of practices that make up a serious application. Each one of these processes, and each one of the other *types* of processes (databases, subroutines, user interface, and so on), forms a different structure; but all exist at the same time, and all use the same software elements. An application *is*, in effect, these processes; it consists of countless structures, all interacting, and the main structure is merely one of them. The idea that an application can be depicted as a perfect hierarchical structure of independent elements is, therefore, nonsensical. Note that this situation is inevitable: a serious application *must* include many processes, and a process *must* affect different parts of the application. The very purpose of processes is to *relate* – in specific ways, on the basis of certain rules or requirements – various parts of our affairs, and hence various parts of the software application that mirrors these affairs.

An application, thus, is a system of interacting structures, not the simple structure we see in diagrams. The other structures are hidden, although we could describe them too with neat diagrams if we wanted. We would need a different diagram, though, for each structure, and the main difficulty – the task of dealing with many structures together – would remain. Because the structures share the application's elements, we must take into account many structures, and also many interactions, at the same time. And it is only our minds – our knowledge and experience – that can do this, because only minds can process complex structures.

❖

It is interesting to compare this system of structures with the system of structures created by *language*. Think of a story describing a number of people, their life and physical environment, their knowledge and activities, their desires and fears. We may consider the main structure of the story to be the structure of linguistic entities, so the story can be viewed as a hierarchy of paragraphs, sentences, and words. In addition, we can view each sentence as a hierarchy of grammatical entities. And if we take into account the *meaning* of the words, we can discern many other structures. Their meaning reveals such entities as persons, objects, and events, all of which have a number of attributes. These entities are related in the story, so for each attribute there exists a structure that relates in a particular way the *linguistic* entities, in order to reflect the relations between the *real* entities.

The fact that we can understand the story proves that the author successfully conveyed to us the relations between persons, objects, and events, permitting

us to discover those structures; for, without those structures, all we would see is the *linguistic* relations. But the structures share their elements – they use the same words, phrases, and sentences; so the story is a system of interacting structures, although only the linguistic ones are manifest. We can discover from the story such structures as the feelings that people have toward one another, or their family relationships, or the disposition of objects in a room, or the sequence of events; and these structures do not parallel the linguistic structure.

Each structure in the story is, simply, one particular way of viewing it, one of its aspects. If we view the story from one narrow perspective, if we interpret, relate, and analyze its elements to reflect one aspect while ignoring all others, we end up with one of these structures. But, clearly, all structures exist at the same time. In the case of language, then, we have no difficulty understanding why the same elements are used in several structures simultaneously. No one (apart from misguided linguists) would claim that the only thing that defines a story is its structure of linguistic entities, or its structure of grammatical entities. So why do we expect a *software* system to be completely defined by its main structure? In software applications, as in stories, the elements are always connected through diverse relations – relations that are not part of the main structure and cannot be predicted from it. This is not an accident, nor a sign of bad programming or writing, but the very nature of these systems. It is precisely this quality that makes those symbols, when processed by a human mind, a meaningful story or software application, rather than a collection of independent structures.

3

Let us examine next the software principle known as *database*, which also gives rise to complex relations between software entities.[2] Databases are the means through which software applications use data, especially the large amounts of data stored in external devices like disks. Data records are read, written, modified, and deleted in many places in the application. And an application may use hundreds of files, and millions of records, connected through intricate relations and accessed through thousands of operations.

The purpose of databases is to relate the various data entities in ways that mirror the relations connecting the *real* entities – those entities that make

[2] The term "database" refers to any set of logically related files, not just those managed formally through a database system. And consequently, the term "database operations" includes not just the high-level operations of a database system, but also the traditional file operations. These terms are discussed in greater detail in chapter 7 (see pp. 686–687).

up our affairs. Database theories encourage us to describe with elaborate definitions and diagrams the data structures; that is, the relations formed by files, records, and fields. And this is a fairly easy task, because most of these relations can indeed be designed (individually, at least) as simple hierarchical structures. But the theories ignore the relations created at the same time by the database *operations*. These operations access the database from different elements of the application, and therefore relate these elements logically.[3]

As is the case with the other types of processes, the effect of database operations can be represented with hierarchical structures. To discover one of these structures, all we need to do is view the application from the perspective of one particular field, record, or file; that is, classify the application's elements according to the operations performed with that field, record, or file, while ignoring their other functions. In most applications we can find a large number of such structures, all using the same elements, and thus interacting with one another and with the structures representing the other processes.

It is impossible for database operations *not* to create structures that relate diverse elements. The very essence of the database principle is to connect and relate various parts of the application by means of data that is stored in files and indexes. We seldom access a set of related files in only one place in the application: we modify fields in one place and read them elsewhere; we add records in one place and delete them elsewhere; we interpret the same records on the basis of one index in one place and of another index elsewhere.

For example, if the program stores a certain value in a database field in one place and makes decisions based on that value in other places, all these places will necessarily be linked logically. They form a structure that, very likely, does not parallel the main structure, nor one of the structures formed by the other processes. For, could this particular relation be expressed through another structure, we wouldn't need that database field to begin with. We need a number of structures precisely because we need to relate the various parts of the application in several ways at the same time. The data stored in the database, together with the operations that access it, provides the means to implement some of these relations. And, just as is the case with the other processes, the structures formed by database operations are seldom manifest;

[3] Although *individually* the file relationships are simple hierarchical structures, a file is usually related to *several* other files, through the same fields or through different fields; and these relationships can seldom be depicted as one *within* another. The totality of file relationships in the database, therefore, is not one structure but a system of interacting structures. This fact is obvious (for instance, if we tried to depict with one hierarchy all the file relationships in a complex application, we would find it an impossible task), so I will not dwell on it. I will only discuss here the structures generated by the database operations, which are less obvious.

we only see them if we separate in our imagination each structure – each set of related elements – from the other structures that make up the application.

Note also that databases merely create on a large scale the kind of relations that memory variables – any piece of storage, in fact – create on a smaller scale. Each variable we use in the application gives rise to a structure – the structure formed by the set of elements that modify or read that variable. And these structures interact, because most elements use more than one variable. Moreover, each element is part of other processes too (part of a business practice, for instance), so the structures created by memory variables also interact with the other *types* of structures.

Thus, let us call *shared data* the broader software principle; namely, all processes based on the fact that the same piece of storage – a database field as well as a memory variable – can be accessed from different elements in the application.

4

Let us study, lastly, the software principle known as *subroutine*. A subroutine is a piece of software that is used (or "called") in several places in the application. Subroutines are software modules, but their ability to perform the same operations in different contexts endows them with additional qualities. (The software entities known as subprograms, functions, procedures, and objects are all, in effect, subroutines.)

We already saw how the practices implemented in the application, as well as the operations performed with databases and memory variables, create multiple, simultaneous structures. But it is with subroutines that the delusion of the main structure is most evident, because subroutines serve both as elements of the main structure and as means of relating *other* elements. The software theorists acknowledge one function, but not the other, so they fail to appreciate the complex role that subroutines play in the application.

The programming theories praise the concept of modularity and the notion of structured, top-down design. Since a module can call other modules, and those can call others yet, and so on, it is possible to create very large hierarchical structures of modules. Thus, the theories say, applications of any size and complexity can be developed by breaking them down into smaller and smaller modules, until reaching modules simple enough to program directly. The application's main structure will then be a hierarchical block diagram where the blocks are the modules, and the branches are the relations between the calling and the called modules. This diagram will represent accurately the application's performance.

We already know why the theories are wrong: the modules are related, not only through the main structure, but also through the structures formed by practices and shared data. And when some of the modules are subroutines, there are even more structures. If a module is used in several places, those places will be linked logically – because they will perform, by way of the shared module, the same operations. It is quite silly to start by deliberately designing two software elements in such a way that they can share a subroutine, and to end by claiming that they are independent; and yet this is exactly what the theories ask us to do. The very fact that a set of operations is useful for both elements demonstrates the existence of a strong logical link between these elements.

So, while *separating* the application into independent elements, subroutines *connect* into logical units *other* parts of the application. Each subroutine, together with its calls, forms a hierarchical structure – a structure that involves some of the application's elements but is different from the main structure, or from the structures formed by the other subroutines, or by the other types of processes. We can picture this structure by viewing the application from the perspective of that subroutine alone, while ignoring its other functions. The structure would represent, for example, the elements that call the subroutine, classifying them according to the ways in which the subroutine affects them.

Only if every module in the application is used *once*, can we say that the sole relations between modules are those depicted by the main structure. When this is true (and if the modules are not related through any other processes), the software theories may well work. But this trivial case can occur only in textbook examples. In real applications most modules are used more than once. As we divide the application into smaller and smaller parts, we are increasingly likely to encounter situations where the same operations are required in different places – situations, that is, where modules become subroutines. This, of course, is why subroutines are so common, why they are such an important programming expedient. The theories are right to encourage modularity; their mistake is to ignore the structures created by the subroutines.

❖

Subroutines are a special case of a broader software principle – a principle that includes all the means of performing a given operation in different places in the application. Consider, for instance, the operation of incrementing a counter, or the operation of comparing two variables, or the operation of adding a value to a total. (It doesn't have to be the *same* counter, or variables, or total; what is shared is the *idea*, or *method*, of incrementing counters, of comparing variables,

of updating totals.) We don't think of these operations as subroutines, but they play, by means of individual statements, the same role that subroutines play by means of modules: they perform the same operation in different contexts. Thus, we can implement these operations as subroutines if we want, but this is rarely beneficial. (Some do become subroutines, in fact, when translated by the compiler into a lower-level language.) Whether physically repeated, though, or implemented as subroutines and called where needed, we have no difficulty understanding that what we have is one operation performed in several places. So, *in our mind*, we are already connecting the application's elements into various logical structures, a different structure for each operation.

We must also include in this broader principle those subroutines that are implemented *outside* the application; for example, those found in subroutine libraries. Let us call them *external* subroutines, to distinguish them from those belonging to the application. The external subroutines are themselves modules, of course, so they may be quite large, and may invoke other modules in their turn. But all we see in the application is their name, so from the perspective of the application they are like the simple operations we discussed previously. Also like those operations, an external subroutine will relate the calling elements logically if called more than once, giving rise to a structure. (Their similarity to simple operations becomes even clearer when we recall that functions available only through external subroutines in one language may well be available through built-in operations in another, specialized language.)

Let us call *shared operations*, therefore, the broader software principle; namely, all processes based on the fact that the same set of operations can be performed by different elements in the application.

Since each process gives rise to a structure, the application is not just the elements themselves, but also the structures formed by these elements, and the interactions between these structures. If the structures represent various aspects of the application, if their purpose is to implement certain relations that exist between the application's elements in addition to the relations established by the main structure, then the interactions simply reflect the simultaneous occurrence of these relations. To put this differently, the complex structure that is the application cannot be approximated with one structure – not even the main structure – because the links between the individual structures are too strong to be ignored. Since each structure is an important aspect of the application, if we ignore the links caused by just one structure the application is bound to be inadequate in at least one class of situations.

The fallacy that all software theories suffer from is the notion that software modules are independent elements: it is assumed that their internal operations are strictly local, and that they are related to other modules only through their input and output. Their internal operations are indeed hidden from the other

modules, but only if judged from a certain perspective (from the perspective of the flow of execution, for instance). Even a module that is invoked only once is rarely independent – because it is usually related to other modules, not just through its input and output, but also through the various processes of which it is part (business practices, for instance).

But the belief in module independence is especially naive for modules that are shared, because, in addition to the relations caused by various processes, their very sharing gives rise to a set of relations. The structure created by calling a subroutine in different places is a reflection of a requirement, of a logical link between these places – a link that must exist if the application is to do what we want it to do. This link is the very reason we use a piece of software that can be invoked in several places. We *need* to relate these places logically; for, if we didn't, we would be using only modules that are invoked once. (And it is merely a matter of interpretation whether the subroutine itself is independent and the elements that call it are not, because related logically through it, or the subroutine too is part of the structure and hence related logically to these elements.) Ultimately, we need the principle of shared modules for the same reason we need the other software principles: to relate the application's elements in many different ways at the same time.

Note that it is not by being one *physical* entity that a subroutine relates the elements that call it. What matters is the *logical* link, so even if we made copies of the subroutine – each call referring then to a different physical entity – the logical link, and hence the structure, would remain. (This is what happens, in essence, when we repeat individual operations – those small pieces of software we decide not to turn into subroutines.)

5

Recall Wittgenstein's example of games. We concluded that games cannot be correctly represented with one hierarchical structure, because it is impossible to capture in one classification all the attributes that games can possess. Instead, we need several structures, one for each attribute, and it is only this *system* of structures that completely represents the concept of games. In other words, there are many structures with the concept of games as the top element and the individual games as the terminal elements; and all these structures exist at the same time. We saw that the only way to account for all the attributes in one structure is by *repeating* some of the attributes at the intermediate levels. But this is no longer a correct hierarchy; moreover, it does not reflect the way we actually perceive games and their attributes. In a correct hierarchy each attribute appears only once.

It is relatively easy to see this problem in hierarchies that depict classifications and categories, but we encounter it in any complex phenomenon. Recall the case of stories. We concluded that a story is a system of interacting structures, where each structure is one particular way of relating the linguistic elements that constitute the story. Each structure, then, depicts one aspect of the story, one attribute. So, if the story is the top element and some small linguistic parts (say, sentences) are the terminal elements, the only correct structures are those that depict its attributes separately. These structures are imaginary, however. Since the top element and the terminal elements are the same in all of them, they cannot exist as separate structures in reality. Thus, to understand the story we must combine them in the mind, and the result is a complex phenomenon: we cannot depict the combination with a simple hierarchical structure. This problem is similar to the problem of depicting in one structure the attributes of games.

For example, referring to one person or another, and referring to one event or another, are attributes of the linguistic elements that make up the story. Consider a trivial story involving only two persons, *P1* and *P2*, and two events, *E1* and *E2*. Figure 4-6 shows the four structures that represent these attributes; and, for simplicity, each attribute has only two values, *Y* and *N*: a given element either is or is not affected by it. Each structure, then, divides the story's sentences into two categories (with no intermediate levels of detail): those that refer to, and those that do not refer to, *P1*; those that refer to, and those that do not refer to, *E1*; and so on. While each combination of two categories includes all the sentences in the story, and is therefore the same for all attributes, the sentences in the individual categories may be different.

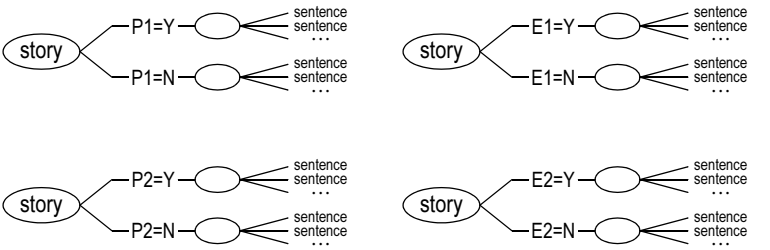

Figure 4-6

Since each sentence possesses all four attributes, these structures interact; so the story is the system comprising all of them. When we read the story, we easily picture such situations as a person involved in two different events, a person involved in neither event, or two persons involved in the same

event. Thus, we can hold in the mind the structures of persons and events simultaneously (because minds can process complex structures), but we cannot represent this phenomenon with one structure.

To combine the four attributes in one structure, we must depict them as *one within another*. For example, we can start with *P1*, include in each of its two branches the two branches of *P2*, and then similarly include *E1* and *E2*. But this is an incorrect hierarchy. Attributes of entities are independent concepts: when an entity possesses several attributes, it possesses them in the same way, not as one within another. The need to repeat attributes, and the fact that we can combine them in any order we like, indicate that this structure does not reflect reality. The only way to depict more than one attribute in one structure is by showing all but the first as *subordinate* to others, while in reality they are independent.

This structure is shown in figure 4-7 (to reduce its size, half of the lower-level elements were omitted). There are sixteen categories of sentences: starting from the top, those that refer to both persons and both events, those that refer to both persons and only to *E1*, those that refer to both persons and only to *E2*, and so on, down to those that refer to neither person and neither event. The diagram clearly demonstrates the need to depict the attributes as one within another – the cause of their repetition. It also demonstrates the multitude of possible combinations: instead of ordering the levels of categories as *P1-P2-E1-E2*, we could order them as *P2-P1-E2-E1*, or as *E1-P1-E2-P2*, etc. While the top element and the terminal elements would be the same, the

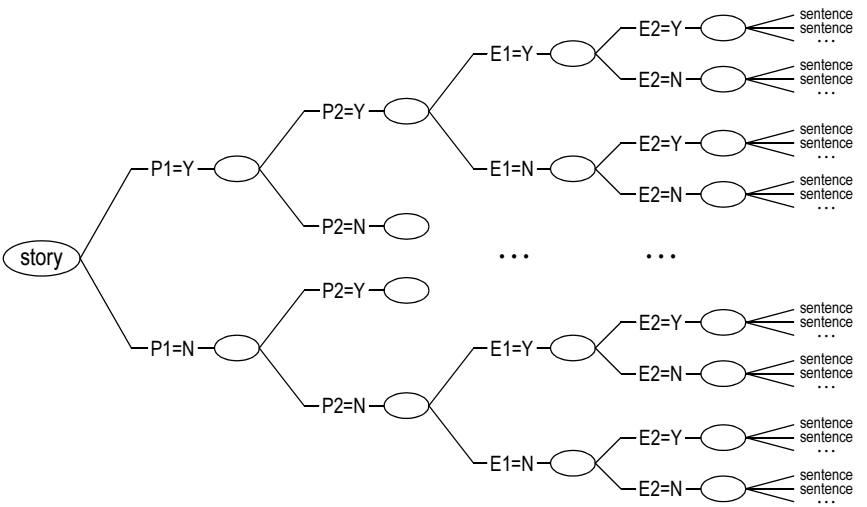

Figure 4-7

intermediate levels would differ. So, when attempting to combine attributes, we end up with several structures depicting the same story. And the absurdity of this situation indicates that these structures do not reflect the actual relationships. In reality, when reading the story, there is only one way to understand the relationships between persons and events.

To appreciate this problem, it may also help to contrast the complex phenomena of games and stories with those situations that *can* be adequately represented with simple structures. A physical structure like the transmission of a car can be described completely and accurately with a hierarchical structure where the individual components are the terminal elements, and the various subassemblies form the levels of abstraction. If, however, we consider such attributes as the colour of the components, or their weight, or the date they were made, or the age of the workers who handle them, we will find that each attribute relates them in a different way. We can view the same transmission, therefore, as a system of many different structures, one structure for each attribute: the top element and the terminal elements are the same in all structures, but the intermediate elements and levels differ. (Levels can be used, for example, to represent increasingly fine details: in the structure depicting the manufacturing date, they may be the year, month, and day; in the structure depicting the colour, they may be light and dark, and various shades; and so on.)

What distinguishes this system of structures from the system that makes up a story is the fact that the structure which concerns us the most – the one depicting the interconnection of components and subassemblies – is very weakly linked to the structures formed by the other attributes. In other words, the other attributes have little or no bearing on our main structure. So, even though the assembly of a transmission gives rise to a complex phenomenon, just like a story or the concept of games, in practice we can ignore the interactions between the main structure and the others. As a result, the hierarchical diagram of components and subassemblies provides an excellent approximation of the whole phenomenon.

Note that if we become interested in other details too – if, for instance, we require a study of that assembly plant, the business of transmission manufacturing, the workers and their life – the phenomenon will change. In the new phenomenon, attributes like a part's manufacturing date or a worker's age are as important as the attributes that affect the operation of the transmission. And consequently, the structures they generate and their interactions can no longer be ignored. Thus, the neat tree diagram depicting the components and subassemblies will no longer provide an adequate approximation of the whole phenomenon.

6

Contrary to the accepted programming theories, software applications are more akin to games and stories than to physical structures like car transmissions. Thus, software applications cannot be correctly represented with only one hierarchical structure. The software entities that make up an application – entities as small as individual operations and as large as entire modules – are affected by various processes; in particular, they call subroutines, use memory variables and database fields, and are part of practices. Since a process usually affects several entities, it serves also to relate them. It endows the entities, therefore, with an attribute, and the attribute creates a hierarchical structure in the application. So there are as many structures as there are processes. We see these structures when we study the application from the perspective of one subroutine, or one memory variable, or one database field, or one practice.

Thus, there is no hierarchy where the entity *application* is the top element, and the individual statements or operations are the terminal elements – not if we want to depict *all* the attributes possessed by these elements. Instead, there are *many* such structures, all using these elements. And it is only this system of simultaneous, interacting structures that completely represents the application.

As in the case of games and stories, the only way to represent all the attributes with one structure is by *repeating* some of them throughout the intermediate levels; and this will not be a correct hierarchy. For instance, if the calling of a particular subroutine is one attribute, then in order to include it in the same structure as another attribute we must repeat it at the intermediate levels. The structure will *look* like a hierarchy, but we know that this was achieved by showing several times what is in fact one attribute. It is impossible to indicate which elements call the subroutine and which ones do not, without this repetition (impossible, that is, if the structure is to include also the other processes implemented in the application – the other subroutines, the business practices, and so forth). The only way to show the subroutine only once is with its own, separate structure. But the terminal elements in this structure are software elements used also by the other processes, so this structure cannot exist on its own, separately from the others; they form the application together.

We encounter the same problem, of course, for any other process. For any one attribute, the only way to prevent its repetition is by creating its own structure. Only by showing the application's elements from the perspective of one attribute and ignoring their other functions can we have a diagram where the attribute appears only once. If we want to represent the entire application

with one structure, then all attributes but one must be repeated, and shown as *subordinate* to others, in order to account for the combinations of attributes. But the attributes are not related in this manner in the actual application, so this structure does not represent it correctly. When a software element possesses several attributes, they are all possessed in the same way, not as one within another.

This problem is illustrated in figure 4-8. As with the structures of games and stories we studied previously, in order to emphasize the problem I am showing a trivial, purely hypothetical situation. The only purpose of this example is to demonstrate that even simple applications give rise to complex structures.

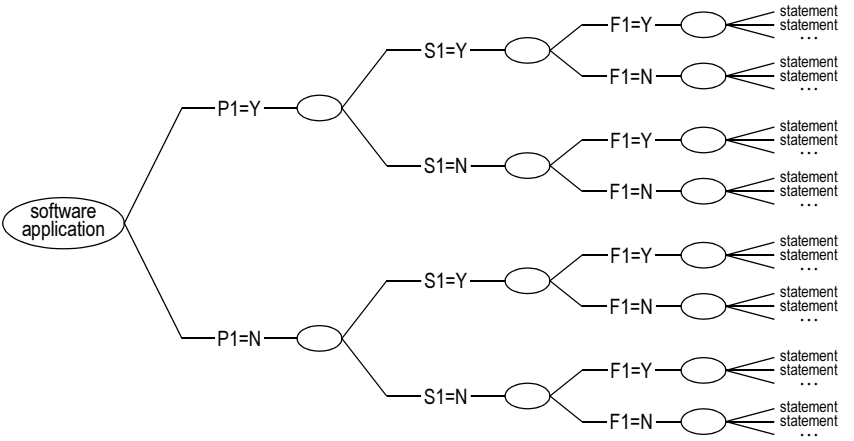

Figure 4-8

There are only three processes, and hence three attributes: *P1* is a particular practice implemented in the application, *S1* is the calling of a particular subroutine, and *F1* is the use of a particular database field. Moreover, each attribute has only two values, *Y* and *N*: a given software element either is or is not affected by a process. In other words, in this simple application there is only one way to be part of that practice (which is unrealistic, of course, even for a simple application), one way to call that subroutine (there are no parameters or returned values, for instance), and one way to use that field (its value can only be read, for instance). The terminal elements of this structure are some small software parts, say, statements. So there are eight categories of statements: those that are part of the practice, call the subroutine, and use the field; those that are part of the practice, call the subroutine, and do not use the field; those that are part of the practice, do not call the subroutine, and use the field; and so on.

The diagram clearly shows why two of the attributes, *S1* and *F1*, must be repeated: if we start with *P1*, the structure can be said to represent the application from the perspective of *P1*; then, the only way to include the other attributes is by depicting each one *within* the categories created by the previous attributes. And it is just as clear that we could draw the diagram by showing the attributes in a different order: we could start with *F1*, for example, and follow with *P1* and *S1*; *F1* would no longer be repeated, but *P1* and *S1* would. The fact that some attributes must be repeated, and that the same application can be represented with different structures, indicates that these structures do not reflect reality.

Figure 4-9 shows the structures that represent the three attributes separately. Like the separate structures of games in figure 4-5 (p. 341), or those of stories in figure 4-6 (p. 357), these *are* correct hierarchies. But, like the others, these structures are imaginary: since the top element and the terminal elements are the same in all three, they cannot actually exist separately. Each statement possesses the three attributes, so the structures interact. It is their combination, a complex structure, that constitutes the application. And we are able to create applications for the same reason we are able to understand stories or the concept of games: because our mind can process complex structures.

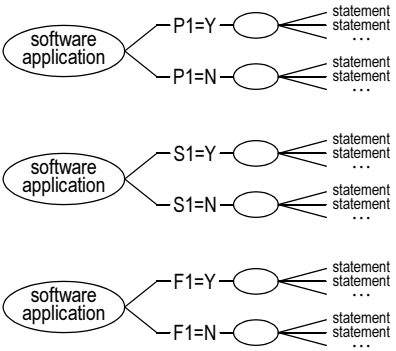

Figure 4-9

Note that there are no intermediate levels in these structures; that is, no intermediate categories of statements. This is due to the assumption that the attributes have only two values, *Y* and *N*. In a real application, most attributes generate a structure with several levels of detail: categories within categories, reflecting the many ways in which a process can affect a statement, or the many reasons it does not affect a statement. But even with the additional levels, these structures remain correct hierarchies if they continue to represent individual

attributes. It is only the attempt to combine all the attributes in one structure that is wrong.

It is obvious that any application can be represented with a classification-style diagram like the one in figure 4-8. And, while a real application may have thousands of attributes and elements, the problem would be the same: since all attributes but one must be repeated, the diagram would not reflect reality. Only a system of structures like those in figure 4-9 represents the application correctly. Thus, by showing that one structure is inadequate, classification-style diagrams remind us that applications consist of interacting structures; so they help us to understand the nature of software. We don't have to actually draw the diagrams, of course; we only need to know that such diagrams exist.

In contrast, diagrams like flowcharts represent only *one* aspect of the application; for example, the flow of execution, or the logic of a particular business practice. By emphasizing one structure and obscuring the others, these diagrams lead to the delusion that applications can be reduced to one structure, the *main* structure; the other structures, we think, can be ignored, or eliminated, or handled separately. With classification-style diagrams it is more difficult to make this mistake, because in these diagrams all structures look alike: since we can draw the diagram with the attributes in any order, it is obvious that *all* structures are important.

The delusion of the main structure is demonstrated by the naive theory known as structured programming. This theory holds that programs can be designed as a strict hierarchical structure of software elements, and gives us methods whereby, supposedly, any piece of software can be transformed into such a structure. But the theory looks only at the *flow-control* structures, ignoring the *other* structures formed by the same elements. Moreover, the recommended transformations do not *eliminate* the unwanted flow-control structures, but merely replace them with structures based on shared data or shared operations. The purpose of the new structures, therefore, is to relate the program's elements in various ways; in other words, to restore, by different means, the links originally provided by the flow-control structures.

What the transformations do, then, is replace the original complex structure, which has *many* flow-control structures plus *many* structures of other types, with a different complex structure, which has *one* flow-control structure plus *even more* structures of other types. If we study only the flow-control type, the program does indeed look now like a neat hierarchy of software entities. If, however, we also take into account the other types, we find that it consists of many interacting structures, just as before.

Because in the program's flowchart the flow-control structures are obvious while the others are not, the advocates of structured programming believe that the program was reduced to one structure. The fallacy, thus, lies in the belief that transformations which reduce the *flow-control* structures to one structure reduce the program itself to one structure. (Actually, the fallacy of structured programming is even greater: as we will see in chapter 7, the flow-control structure itself remains a system of interacting structures; besides, the transformations are rarely practical.)

7

We wish to represent our software applications with neat tree diagrams, and we wish to have software entities that are independent of one another. We appreciate the benefits of these two principles, but we fail to recognize their impossibility in practice. These principles are related, since both are necessary in order to implement simple hierarchical structures. We apply them successfully in our manufacturing and construction activities, where we create large hierarchical structures of independent *physical* entities; and we try to create software applications by applying these principles to *software* entities.

Software, however, like language, is different. The *physical* structures we design and build are our invention, so we deliberately restrict ourselves to simple hierarchical structures in order to ensure their success. With software and language, on the other hand, we want to mirror facts, processes, and events that *already exist* in the world, and which can only be represented with *complex* structures. If we restricted ourselves to simple hierarchical structures, we would be able to represent in software and in language only the simplest phenomena, or we would have to tolerate poor approximations of the complex phenomena. Simple structures are inadequate if we want to represent the world as it actually is, if we want to mirror reality.

Software entities, we saw, are independent only when viewed as elements of *one* structure; for instance, the application's main structure, or a particular process. And this independence is illusory, because they are, at the same time, elements in *other* structures. We can treat them as independent entities, and as elements of a simple structure, only if we ignore the roles they play in the other structures, and the interactions that these multiple roles give rise to. The only way to attain independence of software entities is by ensuring that each entity is used in only one structure; but then our applications would no longer mirror reality.

I want to emphasize again that the complex structures we generate with our software applications are inevitable. The interactions that occur between

software elements are not necessarily a sign of bad programming; we must expect them even in well-designed applications. The software mechanists tell us that we need their theories and programming aids in order to reduce these interactions, but now we understand why these theories and aids cannot help us: if we accept them and restrict our applications to isolated structures of software entities, the applications will not reflect correctly our affairs.

The complex relations that arise in our applications are, thus, an important *quality* of software. Their existence demonstrates that, like language, software *can* have interacting structures, and *can* mirror the world accurately. Instead of trying to *avoid* these relations, then, we must increase our programming expertise so that we can successfully deal with them. We *must* create software structures that share their elements, because this is the only way to mirror, in software, phenomena which themselves consist of structures that share their elements.

In conclusion, the versatility of software is due to the ability to relate the elements of an application in various ways by using the same files, variables, operations, and subroutines in different elements. We need this ability in order to implement processes, and we need processes in order to represent the world. It is the programmer's task to create, for each process, the structure that relates the application's elements in the manner required by that process. And he must do this with many processes at the same time, because this is how the actual facts are related in the world. The very purpose of practices, and of shared data and operations, is to mirror in software those aspects of our affairs that *already* form different structures with the same elements in the real world. So we must not be upset, but pleased, when we end up with interacting structures in software.

If software involves complex structures, software development requires human minds. How, then, can the programming theories and aids help us? Being mechanistic systems, they can only represent *isolated* software structures. So, just to consider them, we must commit the fallacy of reification: we must separate the countless processes that make up an application. The main structure, the practices embodied in the application, the database, are held to represent independent logical structures – structures that can be studied and programmed separately. The reason we are asked to reify these structures is that, within each structure, the theories and aids can offer us the means to start from higher levels of abstraction. Thus, we are also tempted to commit the fallacy of abstraction. Less programming and lower programming skills are needed when starting from higher levels; and this, ultimately, is seen as the chief benefit of programming tools, database systems, and development environments. But when committing the two fallacies, we lose most of the interactions; so our applications will not represent the world accurately.

❖

To recognize the absurdity of programming theories and aids, all we need to do is imagine a similar situation for language. The equivalent of the software industry, and the theories and devices that act as substitutes for programming expertise, would be a *language* industry. Instead of learning how to use language to acquire and to express knowledge, we would learn how to use various systems provided by language companies as substitutes for linguistic competence. And instead of simply communicating through language, we would spend most of our time and resources assimilating an endless series of linguistic innovations – new languages, theories, methodologies, and devices. The equivalent of software development tools and aids would be linguistic tools and aids that promise higher levels of abstraction. These devices would extract individual structures from the complex phenomenon of language – the syntax of a sentence, the logic of an argument, the aspects of a story – addressing each one in isolation. Within each structure, the devices may even do what they promise; but this would not help us to use language, because language structures interact, and we need the ability to deal with all of them simultaneously.

The only thing that language companies could offer us, then, is a way to start from higher levels of abstraction within each structure. We would have to use a set of ready-made sentences and ideas, for example, instead of creating our own, starting with words. This would perhaps expedite the generation of individual structures, but at the cost of reducing the number of alternatives for the concepts we represent with language. The interactions between language structures occur not only at the level of sentences and ideas, but also at the low level of words: it is the meaning of individual words that usually determines the attributes of linguistic elements – attributes which ultimately cause the interactions. If we could no longer express ourselves starting with words, many interactions would become impossible – not only between language structures, but also between the language structures and other knowledge structures.

What this means in practice is that many ideas, or stories, or kinds of knowledge, or forms of discourse, would also become impossible. The consequence of linguistic reification and abstraction, thus, is not just an impoverishment in language structures and in the kind of concepts that can be represented with language. Because these concepts are linked with all human knowledge, our entire existence would be impoverished. Other knowledge structures we hold in our minds – our thoughts, feelings, beliefs, and expectations, which interact with the linguistic structures – would be restricted to a small number of alternatives.

And so it is with programming. When we start from higher levels of abstraction, and when we separate the various aspects of an application, we end up with impoverished software; that is, software which represents only a fraction of the possible alternatives. We *can* mirror our affairs in software, but only if we start from low levels. We cannot if we rely on mechanistic programming theories and on ready-made pieces of software.

What is worse, as software is acquiring a social role similar to that of language, an impoverishment in the alternatives that we represent with software will affect us just as it would for language: other knowledge structures we hold in our minds – structures that appear unrelated to software but interact, in fact, with the software structures – will be impoverished at the same time. This threat is what we will examine in the next two chapters.