Patterns of Software

Tales from the Software Community

Richard P. Gabriel

New York Oxford
OXFORD UNIVERSITY PRESS
1996

Oxford University Press

Oxford New York
Athens Auckland Bangkok Bogota
Bombay Buenos Aires Calcutta Cape Town
Dar es Salaam Delhi Florence Hong Kong Istanbul
Karachi Kuala Lumpur Madras Madrid
Melbourne Mexico City Nairobi Paris
Singapore Taipei Tokyo Toronto

and associated companies in
Berlin Ibaden

Published by Oxford University Press, Inc.,
198 Madison Avenue, New York, New York, 10016-4314

Library of Congress Cataloging-in-Publication Data
Gabriel, Richard P.
Patterns of software: tales from the software community.
p. cm. Includes Bibliographical references
ISBN 0-19-5100269-X
1. Computer software—Development. 2. Object-oriented
programming (Computer science) I. Title.
QA76.76.D47G33 1996
005. 1—dc 2095-41883

1 3 4 5 7 9 8 6 4 2

Printed in the United States of America
on acid-free paper

To Jo Lawless, who led me away from the lion

Midway on our life's journey, I found myself
In dark woods, the right road lost. To tell
About those woods is hard—so tangled and
rough

And savage that thinking of it now, I feel
the old fear stirring: death is hardly more bitter.

Preface

The essays in this book started out as a series of columns for the *Journal of Object-Oriented Programming*. I was trying to model myself somewhat after Samuel Johnson, and the series was aimed at being the digital age's equivalent of *The Rambler*. I'm certain I didn't succeed in matching Johnson's wit and style, but I matched at least two of his characteristics—procrastination and laziness. Johnson was well known for writing his essays right up to the deadline, often keeping the publisher's runner waiting for the manuscript as Johnson completed it. In fact, you can notice the effect in many of his essays: An essay starts to make an argument in one direction (corresponding to the first sheets Johnson handed the runner) and then the argument shifts radically or even to the opposite pole as Johnson continued writing and thinking—but revision of the earlier parts was impossible, as it was being typeset for final copy as Johnson pondered.

For my essays I chose the stance of *critic at large*. In this role I attempted to examine topics of interest to the object-oriented programming community from the point of view of someone whose experience has always had a sour component—this will be the subject of some of the essays. I had been working in the object-oriented field for seven years when I started writing them.

Object-oriented programming has a very long history, which the newly initiated sometimes finds surprising. First designed primarily as a means of simulation, later as programming languages for children, object-oriented languages have become nearly mainstream, largely as a means of achieving reuse and productivity.

You see, the history of computing is replete with attempts at making programming easier and, more important, cheaper. Current software makes machines of a sort previously unknown and with capabilities unheard of, partly because there can be built into them some small portion of reason and consideration and intelligent—rather than merely mechanical—reaction. Automobiles can run for dozens of thousands of miles without tuning because their internal operation is governed by software—it's almost more likely that your car can be fixed by a Ph.D. in computer science than by your grandfather's auto mechanic.

When we start cataloging the gains in tools sitting on a computer, the benefits of software are amazing. But, if the benefits of software are so great, why do we worry about making it easier—don't the ends pay for the means? We worry because making such software is extraordinarily hard and almost no one can do it—the detail is exhausting, the creativity required is extreme, the hours of failure upon failure requiring patience and persistence would tax anyone claiming to be sane. Yet we require that people with such characteristics be found and employed and employed cheaply.

We've tried to make programming easier, with abstraction as a tool, with higher-level programming languages, faster computers, design methodologies, with rules of thumb and courses and apprenticeships and mentoring, with automatic programming and artificial intelligence. Compilers, debuggers, editors, programming environments. With structured programming and architectural innovations.

With object-oriented programming.

But programming still requires people to work both alone and in teams, and when people are required to think in order to achieve, inherent limitations rule. Object-oriented programming—which is merely a set of concepts and programming languages to support those concepts—cannot remove the need to think hard and to plan things, to be creative and to overcome failures and obstacles, to find a way to work together when the ego says not to, that the failures are too many and too pervasive.

Reuse, productivity, reliability—these are values prized by managers and moneymakers.

Software creators are usually called *engineers*, a connotation that usually brings to mind a person who applies well-known principles and methods to create variations on known themes. For example, a bridge builder is an engineer who uses a long list of known techniques to erect a structure to traverse rivers, chasms, and rough terrain while supporting a certain load while withstanding natural forces like wind and earthquakes.

Even though the word engineer comes from the same basic root as ingenuity does, the feeling one gets when hearing the word is of careful, detailed, nearly plodding predictability of work. This makes sense to a degree because engineering disciplines frequently have handbooks from which they work that prescribe a series of rules and principles and ways of solving problems according to a long tradition and experience. For instance, bridge builders have centuries of experience bridge building to draw upon.

Building software—some call it *software engineering*—is only 30 or 40 years old, and it shares with other engineering disciplines virtually nothing. Engineering teams for bridge building are composed of well-known roles whereas in software we are still experimenting. While building bridges, known solutions are adapted to the situation at hand whereas in software we frequently need to invent new techniques and technology. What's easy and hard is not known, and there are very few physical principles to guide and constrain us.

To emphasize this, consider that not only is there a large set of principles for bridge building but there are hundreds of known examples of bridges and even we, as laypeople, know of some of the best examples and even one of the worst—the Tacoma Narrows Bridge, which catastrophically collapsed 50 years ago. But in software, there isn't even a literature of programs that programmers know and can talk about.

The Tacoma Narrows Bridge—let's think about it for a minute. This was a bridge built in Washington State across a strait in the 1940s. Because of the gorge it spanned, it was subject to strong winds. The engineers who built it adapted a design used on the East Coast that was known to be able to withstand strong wind. However, the designers added a feature for pedestrians, a low windbreak about the height of a guardrail that would protect people and cars from the wind. But as soon as this fence was built, the bridge started oscillating from the wind, which flowed over it like an airfoil.

After a few months on a day when the wind was particularly strong and at a particular speed, the airfoil started to oscillate wildly, and the bridge collapsed. The incident was captured on newsreels. The only casualty was a dog who would not get out of a car when its master tried to coax him out to safety.

Even though it was a disaster, the methodology was to modify an existing solution, and when it failed its failure was analyzed. How often does that happen in software? Almost never, because such failures are simply locked away and forgotten—perhaps the folks who participated learn something, but the project is rarely viewed by people outside the project, and there is little interest in the failure itself except perhaps because of its effects on the organization that sponsored it.

So yes, we are engineers in the sense of using cleverness and inventiveness to create an artful engine that is itself clever. But we don't have—perhaps yet—the inherent predictability of schedules and results to be engineers in the sense most people, especially businessfolk, expect.

One of my goals in writing these essays was to bring out the reality of commercial software development and to help people realize that right now software development—except when a project essentially is creating a near variant of an existing program—is in a state where the artifact desired is brand new and its construction is unknown, and therefore the means to approach its construction is unknown and possibly difficult to ascertain; and, furthermore, a group of people is trying to work together—maybe for the first time—to accomplish it. An image I like to use is that every large software project is similar to the first attempt to build a flying-buttress construction cathedral. Imagine how many of them collapsed before we figured out how to build them.

Software development is done by people with human concerns; although someday this component will be a smaller part of the total picture, today it is the high-order bit. The software community approaches these issues with high hopes and a pride in the term engineer. I approach it as a critic.

Let me turn to what I hoped to accomplish as a critic at large. I start with a quote from Samuel Johnson's *The Idler*:

> *Criticism is a study by which men grow important and formidable at a very small expence. The power of invention has been conferred by nature upon few, and the labour of learning those sciences which may by mere labour be obtained is too great to be willingly endured; but every man can exert such judgement as he has upon the works of others; and he whom nature has made weak, and idleness keeps ignorant, may yet support his vanity by the name of a Critick.* (Number 60, Saturday, June 9, 1759)

A critic, at his best, aims at raising questions that otherwise might remain hidden. The role of a critic is to look at things in new ways, to present a perspective that others with less time on their hands can use as the basis for real progress, to ask the question that turns the course of inquiry from a backwater whirlpool toward rapids of exciting new work.

So don't look to these essays for answers—I planned not to, and yet dare not, claim new insight or wisdom: You have to find that for yourself or within yourself. I claim only to be able to expand the playing field to include new considerations and maybe new perspectives.

However, be warned: Nothing is off limits for me. There are no dead ends I won't go down, no agreements I'll honor about what's on or off limits. Every idea out there is fair game—yours included.

But don't worry that your pet idea will be abused or harmed—again Johnson:

> *This profession has one recommendation peculiar to itself, that it gives vent to malignity without real mischief. No genius was ever blasted by the breath of criticks. The poison which, if confined, would have burst the heart, fumes away in empty hisses, and malice is set at ease with very little danger to merit. The Critick is the only man whose triumph is without another's pain, and whose greatness does not rise upon another's ruin.*

The bias I started with in these essays was this:

The promise of object-oriented programming—and of programming languages themselves—has yet to be fulfilled. That promise is to make plain to computers and to other programmers the communication of the computational intentions of a programmer or a team of programmers, throughout the long and change-plagued life of the program. The failure of programming languages to do this is the result of a variety of failures of some of us as researchers and the rest of us as practitioners to take seriously the needs of people in programming rather than the needs of the computer and the compiler writer. To some degree, this failure can be attributed to a failure of the design methodologies we have used to guide our design of languages, and to larger degree it is due to our failure to take seriously the needs of the programmer and maintainer in caretaking the code for a large system over its life cycle.

This book is broken into five parts:

- Patterns of Software
- Languages
- What We Do
- Life of the Critic
- Into the Ground

"Patterns of Software" explores the work of the architect Christopher Alexander as it relates to the creation of software. Christopher Alexander has spent the bulk of his professional life—from around 1960 through the mid-1990s—trying to find the way that those who build buildings, cities, and towns also create beauty and what Alexander calls the *quality without a name*.

Over the last decade, the computer software community discovered Alexander and his concept of *pattern languages* and has tried to incorporate those ideas into software design. The essays in this part examine Alexander's quest for the quality without a name and for beauty and try to see what connections to software, especially object-oriented software, are appropriate.

"Languages" looks at programming languages and how software developers use and react to them. The choice of a programming language seems as sacred and personal as the choice of religion, and that choice often favors languages whose performance characteristics match computer architecture capabilities rather than the capabilities of the people who use programming languages.

"What We Do" sketches the activities we as computer scientists and software developers do and how folks outside our community view us. Here I talk a little bit about writing—which is dear to me—and how our assumptions about the obviousness of the importance of what we do might not be and is not shared by the rest of the world.

"Life of the Critic" is an intellectual autobiography in which I hope to show how I arrived at my views and why I drifted toward the role of critic. I also hope to show that you don't have to start out with a silver spoon in your mouth to be able to make a contribution to humanity—that even someone with as checkered and failure-ridden past as I have can contribute. Many times in my life I despaired when I compared my progress with that of my fellows who never seemed to have had a slip in their careers, and part of my goal, therefore, is to show that someone of average intelligence and talents can do well, even in our modern world.

"Into the Ground" is the story of the company I founded—from its birth in 1984 until its death in 1994. The lessons to be learned from this experience center on the fact that the company carried out its technical agenda perfectly yet it failed miserably, and accompanied by circumstances almost unprecedented in Silicon Valley startup history.

My overall bias is that technology, science, engineering, and company organization are all secondary to the people and human concerns in the endeavor. Companies, ideas, processes, and approaches ultimately fail when humanity is forgotten, ignored, or placed second. Alexander knew this, but his followers in the software pattern language community do not. Computer scientists and developers don't seem to know it, either.

These essays . . . these essays aim to correct that.

I would like to thank Christopher Alexander for his deep, prophetic work on architecture, patterns, and beauty; for writing the Foreword; and for providing original art. Katalin Bende, in Alexander's office, was most gracious in providing assistance with the artwork.

Some essays are reprinted with permission from the Journal of Object-Oriented Programming (copyright © 1991–1993 SIGS Publications, 71 West Twenty-third Street, Third floor, New York, New York 10010). Quotes from *Christopher Alexander: The Search for a New Paradigm in Architecture* by Stephen Grabow were provided by permission from International Thomson Publishing Services Ltd.

Bill Zobrist, Krysia Bebick, and Irene Pavitt of Oxford University Press provided tremendous editorial assistance and support during the preparation of this book.

Mountain View, Calif. R.P.G.

Contents

PART I

PATTERNS OF SOFTWARE

Reuse Versus Compression

Maybe it's my failing memory, but I recall that the hook that grabbed the mainstream world and pulled it toward object-oriented programming was reuse. One of the larger problems that development managers face is how to get big software projects done fast. Most people agree that maintenance is a big part of the overall software problem, but to organizations whose survival depends on getting out new projects or products, the important issue is getting the new software done.

Within each organization writing a lot of software—and among many organizations writing a lot of software—it seems that a lot of that software should be reusable. If this were true, then it would be possible to take some of the code that already exists and put it toward the new project, thereby reducing the time to produce the desired software, Furthermore, if code is reused, it is more likely to be well tested and possibly bug free, and even if it isn't, the maintenance of the various programs that use the reused code should be easier.

Reuse is not a new idea. For decades languages have supported the notion of libraries. A library is a set of subroutines, usually in a particular application area. Old examples are the scientific subroutine libraries for FORTRAN. A similar idea is the Collected Algorithms published by ACM years ago in an ALGOL-like publication language. I remember when I was a kid in 1968 looking up algorithms for sorting and searching in my first programming job.

However, what every manager learns is that reuse under these circumstances requires a process of reuse or at least a policy. First, you need to have a central repository of code. It doesn't help if developers have to go around to other developers to locate code you might be able to use. Some organizations are small enough that the developers can have group meetings to discuss needs and supplies of code.

Second, there has to be a means of locating the right piece of code, which usually requires a good classification scheme. It does no good to have the right piece of code if no one can find it. Classification in the world of books, reports, magazines, and the like is a profession, called *cataloging*. Librarians help people find the book . But few software organizations can afford a software cataloger, let alone a librarian to help find the software for its developers. This is because when a development manager has the choice of hiring another developer or a software librarian, the manager will always hire the developer. It's worth looking at why this is, and I'll return to it later.

Third, there must be good documentation of what each piece of code in the repository does. This includes not only the interface and its purpose but also enough about the innards of the code—its performance and resource use—to enable a developer to use it wisely. A developer must know these other things, for example, in order to meet performance goals. In many cases such documentation is just the code itself, but this information could be better provided by ordinary documentation; but again, a development manager would prefer to hire a developer rather than a documentation person.

For a lot of pieces of code it is just plain simpler to write it yourself than to go through the process of finding and understanding reusable code. Therefore, what development managers have discovered is that the process-oriented world of reuse has too many barriers for effective use.

Looking at the nature of this approach to reuse we can see that it is focused on reusing code from one project to another rather than within a project; the mechanism for reuse is an organization-wide process.

Enter object-oriented programming. The primary point of object-oriented programming is to move the focus of program design from algorithms to data structures. In particular, a data structure is elevated to the point that it can contain its own operations.

But such data structures—called *objects*—often are related to one another in a particular way: One is just like another except for some additions or slight modifications. In this situation, there will be too much duplication if such objects are completely separate, so a means of inheriting code—*methods*—was developed.

Hence we see the claim of reuse in object-oriented languages: When writing a single program, a programmer reuses code already developed by inheriting that code from more general objects or classes. There is a beauty to this sort of reuse: It requires no particular process because it is part of the nature of the language.

In short, *subclassing* is the means of reuse.

Despite this simplified view of reuse, the idea that object-oriented languages have reuse as part of their very essence proved to be a large attraction to the mainstream community. To be sure, there were other attractions as well; here are some:

- Objects and classes of objects are a natural way for programmers to organize programs.

- Systems written with objects and classes are simpler to extend and customize than traditionally constructed ones.

However, the form of reuse in object-oriented languages hardly satisfies the broad goals of software development. What I want to suggest is a better word than *reuse* and maybe a better concept for the reuse-like property of object-oriented languages.

The word (and concept) is *compression*. Compression is the characteristic of a piece of text that the meaning of any part of it is "larger" than that piece has by itself. This is accomplished by the context being rich and each part of the text drawing on that context—each word draws part of its meaning from its surroundings. A familiar example outside programming is poetry whose heavily layered meanings can seem dense because of the multiple images it generates and the way each new image or phrase draws from several of the others. Poetry uses compressed language.

Here is a simple single-sentence example: *My horse was hungry, so I filled a morat with oats and put it on him to eat.*

This sentence is compressed enough that the meaning of the strange word *morat* is clear—it's a feed bag. Pronouns work because of compression.

Compression in object-oriented languages is created when the definition of a subclass bases a great deal of its meaning on the definitions of its superclasses. If you make a subclass of a class which adds one instance variable and two methods for it, the expression of that new class will be simply the new variable, two methods, and a reference to the existing class. To some, the programmer writing this subclass is reusing the code in the superclass, but a better way to look at it is that the programmer is writing a compressed definition, the bulk of whose details are taken from the context in which the superclass already exists.

To see this, note that the subclass definition is frequently some distance from the superclass definition, and so the programmer is relying on knowledge of the superclass to write the shorthand definition.

Compressed code has an interesting property: The more of it you write—the further down the class hierarchy you go—the more compressed the new code becomes. This is good in at least one way: Adding new code is very compactly done. For a development manager this *can* mean that the new code can be written more quickly.

Here is where the change of perspective can help us: Compression has clear disadvantages, and these disadvantages can help explain why object-oriented languages have not solved the software problem (of course, there are many reasons, but this is just the first essay, after all).

Compression is a little dangerous because it requires the programmer to understand a fair amount about the context from which compressed code will take its meaning. Not only does this require available source code or excellent documentation, but the nature of inherited language also forces the programmer to understand the source or documentation. If a programmer needs a lot of context to understand a program he needs to extend, he may make mistake because of misunderstandings.

Furthermore, even if the new code—the compressed code—is compact, it will take at least as much time and effort to write as it would to write the uncompressed code, unless the overall complexity and size of the total code is small or unless the person writing the code has the existing code firmly in mind.

Maintaining compressed code requires understanding its context, which can be difficult. The primary feature for easy maintenance is *locality*: Locality is that characteristic of source code that enables a programmer to understand that source by looking at only a small portion of it. Compressed code doesn't have this property, unless you are using a very fancy programming environment.

Compression has another obvious problem. If you build derived code that tightly depends on some base code, then changing the base code can be expensive and dangerous. If the developer of the base code is not the same person as the developer of derived code, the derived code can be in jeopardy. This is just the problem of all by-reference relationships. Compression in poetry is fine because the ultimate definitions of the words and phrases are outside the poet's mind. Not so for compression in programs: The meanings of the words—the classes—are determined by the programmer. This problem is not unique to class-based compression but applies to any abstraction-based compression.

I don't want to imply that compression is bad—in fact, it is an important resource for incremental definition—but abstractional problems like the risk of changing base code can get in the way.

Compression doesn't help interproject reuse, because to do so requires exporting classes that can be reused. This simply lands us in the classic reuse situation we had earlier, in which the primary barrier to reuse is one of process rather than of technology.

Remember I stated that a development manager will always hire a developer over someone whose job is to effect reuse. Why is this?

Some of the reasons should now be apparent. First, reuse is easiest within a project instead of between them. A manager's success depends on performance on a given project and not on performance over several projects. And preparing code for reuse requires additional work, not only by the reuse expert but also by developers. Therefore, preparing for reuse has a cost for any given project.

Finally, a project's success often depends at least partly on the performance of the code that is developed. The development manager knows that plenty of code to be developed really is different from existing code if for no other reason than the new code has to be specialized to the particular situation to be fast enough. Therefore, in this situation reuse doesn't help very much.

What we've seen is that reuse is properly a process issue, and individual organizations need to decide whether they believe in its long-term benefits. Object-oriented languages provide compression, which can substitute for reuse within a program, but at a cost in maintenance.

Compression is not reuse, and both reuse and compression have their costs and savings.

Habitability and Piecemeal Growth

The sun paints the scattered clouds red, paints the hills across the Bay a deeper red. Moment by moment the sun's light flares windows in the hills, moving upward as twilight approaches. The programmer looks up from his screen to watch. He sees headlights moving one way and taillights the other along either side of the highway spine. Homelights click on, and the Bay is a silhouette. His daughter: Will she be too petrified to dance? Will she miss the cue? If only he could help her pass by her fear. Reaching to the right of the mouse, he lifts his Coke and drains it, saves his buffers, drifts his eyes to the sensual movement of the highway, pausing his mind as it passes over a detail in the code he's just typed. The thought passes before crystallizing, so he puts the CDs in his blue Kelty pack—and heads for the recital.

Code is written by people. And people have a lot on their minds, they have lives to lead, and those lives intrude on their thinking. You might pay a developer for a full eight-hour day, but how much of that day is truly yours? Of course, not all of us are software development managers, but the concerns of these managers should rightfully be concerns of ours—concerns of language designers. The astute software manager knows that just about every programmer except the pathological workaholic trades his hours of overtime for hours of undertime. The manager goes all out to ensure that the needs of the developer are met—home terminals, sunny offices, minimal meetings, meals during the big push, and a healthy ego boost from time to time.

If the software manager knows that people and their habits are the determining factor in software development, why don't language and system designers? Here is what Stroustrup says about the design of C++:

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. (Stroustrup 1987)

This is an interesting idea—no doubt it would be a surprise to most C++ programmers. But, Stroustrup goes on a little more honestly to say:

> *C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency. . . . Features that would incur run-time or memory overheads even when not used were avoided. . . . C++ types and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. . . . They can, however, be used freely without incurring run-time or space overheads.*
> (Stroustrup 1987)

I suppose that the enjoyable part of programming that Stroustrup refers to is the pleasing degree of efficiency that a program acquires when it has no run-time or space overhead.

Like Stroustrup, R. D. Tennent starts out with an admirable goal for programming languages:

> *A programming language is a system of notation for describing computations. A useful programming language must therefore be suited for both* describing *(i.e., for human writers and readers of programs) and for* computation *(i.e., for efficient implementation on computers). But human beings and computers are so different that it is difficult to find notational devices that are well suited to the capabilities of both. Languages that favor humans are termed* high-level, *and those oriented to machines* low-level. (Tennent 1981)

He goes on to describe machine language as the most "powerful" low-level language, but he tellingly remarks:

> *It might be thought that "natural" languages (such as English and French) would be at the other extreme. But, in most fields of science and technology, the formalized notations of mathematics and logic have proved to be indispensable for precise formulation of concepts and principles and for effective reasoning.* (Tennent 1981)

Precise formulation and effective reasoning—again, not exactly what I had in mind. Finally, let's look at Modula-3, a modern, enlightened language:

> *Modula-3 is a modern, general-purpose programming language. It provides excellent support for large, reliable, and maintainable applications. . . . The nature of programming has changed. For many years we were puzzle-solvers, focused on turning algorithms into sets of instructions to be followed by a computer. We enjoyed solving these*

> *puzzles, and we often viewed both the complexity of the puzzle and the obscurity of the solution as evidence of our skill. . . . Aware of our human limitations, we have come to view complexity and obscurity as faults, not challenges. Now we write programs to be read by people, not computers.* (Harbison 1992)

This is the best start, and its finish is not bad either:

> *There is a pleasure in creating well-written, understandable programs. There is a satisfaction in finding a program structure that tames the complexity of an application. We enjoy seeing our algorithms expressed clearly and persuasively. We also profit from our clearly written programs, for they are much more likely to be correct and maintainable than obscure ones.* (Harbison 1992)

Although I think the goal of Modula-3 is not ideal, it is better than the precise formulation goal that Tennent advocates, and it is in a different league from Stroustrup's pleasure of efficiency.

There is, I think, a better goal, to which I want to draw your attention. It's a characteristic of software that you've perhaps not thought of and which perhaps should have some influence over the design of programming languages and certainly of software methodology. It is *habitability.*

Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently. Either there is more to habitability than clarity or the two characteristics are different. Let me talk a little bit more about habitability before I tackle what the difference may be.

Habitability makes a place livable, like home. And this is what we want in software—that developers feel at home, can place their hands on any item without having to think deeply about where it is. It's something like clarity, but clarity is too hard to come by.

Habitability is related to a concept called *organic order* that Christopher Alexander, the architect, uses in his work:

> *Organic Order: . . . the kind of order that is achieved when there is a perfect balance between the needs of the parts and the needs of the whole.* (Alexander 1975)

How are architecture and software related?

I've heard Gregor Kiczales—one of the CLOS designers—say that he wishes that computer science practice could reach the level of engineering excellence that creates buildings like the Superdome in New Orleans. He points out that the design of the Superdome puts together pieces made from a variety of materials and from a range of engineering and building disciplines. The result is a monument to that engineering skill. This is a tempting picture, but I think it's off base.

Buildings like the Superdome lack habitability. In this instance people inhabit the building, but only for very short periods of time, and for very special occasions—and such buildings are not easily grown or altered. The Superdome is a static building, and therefore it can stand as a monument, being little else.

A modern skyscraper, to take another example, has a fixed inflexible interior, which is secondary to the designed beauty of the exterior. Little attention is paid to the natural light, and often the interiors are constructed as "flexible office space," which means cubicles. The flexibility is for management to set up offices for the company, not for the inhabitants—the employees—to tailor their own space. When you run out of space in the skyscraper, you build another; you don't modify the existing one or add to it.

Contrast this with the New England farmhouse. It starts as a small home with a barn out back. As the family grows and the needs of the farm grow, a back room is added to the house, then a canning room, then a room for grandma; stables are added to the barn, then a wing for milking more cows. Finally the house and barn are connected because it is too difficult to get from the house to the barn in a blizzard. The result is rambling, but each part is well-suited to its needs, each part fits well with the others, and the result is beautiful because it is a living structure with living people inside. The inhabitants are able to modify their environment because each part is built according to familiar patterns of design, use, and construction and because those patterns contain the seeds for piecemeal growth.

I think this should be the goal for computer science practice. Most programming languages are excellent for building the program that is a monument to design ingenuity—pleasingly efficient, precise, and clear—but people don't build programs like that. Programs live and grow, and their inhabitants—the programmers—need to work with that program the way the farmer works with the homestead.

This, I think, is the challenge of programming language design in the next generation: to recognize, finally, what programming really is and to address those issues, not issues of elegance and monumental programs.

What are some of the things that contribute to uninhabitable programs? Overuse of abstraction and inappropriate compression come to mind. But that's a topic for another day; today is just to explore the concepts of habitability and piecemeal growth.

In Alexander's definition of organic order applied to software, the concept of "needs of the whole" refers to the grand design or architecture of the piece of software under development, and "needs of the parts" refers to the inevitable changes the various parts of the software undergo. It's difficult to change the grand design of software: You cannot expect to evolve a window system into a spreadsheet.

Although the primary need of the whole is to remain true to its essence, the parts often must change. For instance, one sort of window system could evolve into another.

Software needs to be habitable because it always has to change. Software is subject to unpredictable events: Requirements change because the marketplace changes, competitors change, parts of the design are shown wrong by experience, people learn to use the software in ways not anticipated. Notice that frequently the unpredictable event is about people and society rather than about technical issues. Such unpredictable events lead to the needs of the parts which must be comfortably understood so they can be comfortably changed.

Consider bugs. Many a bug is the result of not anticipating a particular event or use and is not the result of a mistake—bugs are not always errors. Bugs tell us that we are not capable of producing a *master plan*. A master plan is a detailed design, and many projects consider critical their detailed designs. But a master plan is usually not possible, especially for extensive, long-lived software. Alexander writes:

> *It is simply not possible to fix today what the environment should be like [in the future], and then to steer the piecemeal process of development toward that fixed, imaginary world.* (Alexander 1975)

This simply acknowledges that it is impossible to predict the circumstances of a long-lived program. But there is a more important point:

> *Master plans have two additional unhealthy characteristics. To begin with, the existence of a master plan alienates the users. . . . After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect only relatively trivial details. When people lose the sense of responsibility for the environment they live in, and realize that there are merely cogs in someone else's machine, how can they feel any sense of identification with the community, or any sense of purpose there?*
>
> *Second, neither the users nor the key decision makers can visualize the actual implications of the master plan.* (Alexander 1975)

It should be clear that, in our context, a "user" is a programmer who is called upon to maintain or modify software; a user is not (necessarily) the person who uses the software. In Alexander's terminology, a user is an inhabitant. A client or software user certainly does not inhabit the code but instead uses its external interface; such a software user would be more like the city sewer, which hooks up to a building but doesn't live in it.

Several points come to mind. First, when you design software, let the implementers complete those parts of the design for which they are responsible.

Second, have you ever heard a good manager ask the group he or she manages "who owns this?" It's because the manager knows that the excellent employee needs to feel he or she has some authority over what they are responsible for. Further, a well-knit group has this same sense of ownership over what they work on plus a bond of "elitism" that holds them together and makes each member of the team feel responsible for the others' success.

Third, how do you enable a programmer to feel responsible for software developed earlier? Here is where habitability comes in. Just as with a house, you don't have to have built or designed something to feel at home in it. Most people buy houses that have been built and designed by someone else. These homes are habitable because they are designed for habitation by people, and peoples' needs are relatively similar. As I said earlier, a New England farmhouse is habitable, and the new owner feels just as comfortable changing or adapting that farmhouse as the first farmer was. But a home designed by Frank Lloyd Wright—though more habitable than most "overdesigned" homes—cannot be altered because all its parts are too rigidly designed and built. The needs of the whole have overshadowed the needs of the parts and the needs of the inhabitants.

Finally, if Alexander's lesson applies to software, it implies that a development project ought to have less of a plan in place than current thinking allows. This provides a mechanism for motivation and a sense of responsibility to those developers who later must work with the code.

Alexander goes on:

> *The principle of organic order: Planning and construction [implementation, in our context] will be guided by a process which allows the whole to emerge gradually from local acts.* (Alexander 1975)

This just piecemeal growth. Here is how Alexander puts it:

> *[E]ach new building is not a "finished" thing. . . . They are never torn down, never erased; instead they are always embellished, modified, reduced, enlarged, improved. This attitude to the repair of the environment has been commonplace for thousands of years in traditional cultures. We may summarize the point of view behind this attitude in one phrase:* piecemeal growth. (Alexander 1975)

Piecemeal growth is a reality. What gets in its way and prevents software habitability is overdesign, overabstraction, and the beautiful, taut monument of software. Alexander calls this *large lump development*:

> *Large lump development hinges on a view of the environment which is static and discontinuous; piecemeal growth hinges on a view of the environment which is dynamic and continuous. . . . According to the large lump point of view, each act of design or construction is an isolated event which creates an isolated building—"perfect" at the time of its construction, and then abandoned by its builders and designers forever. According to the piecemeal point of view, every environment is changing and growing all the time, in order to keep its use in balance; and the quality of the environment is a kind of semi-stable equilibrium in the flux of time. . . . Large lump development is based on the idea of* replacement. *Piecemeal growth is based on the idea of* repair. (Alexander 1975)

Recall that one of the tenets of encapsulation is that the interface be separate from the implementation because this permits the implementation to be replaced when needed.

The problem with traditional approaches to abstraction and encapsulation is that they aim at complete information hiding. This characteristic anticipates being able to *eliminate* programming from parts of the software development process, those parts contained within module boundaries. As we've seen, though, the need to program is never eliminated because customization, modification, and maintenance are always required—that is, piecemeal growth.

A better goal is to *minimize* or *reduce* the extent of programming, which implies providing mechanisms that allow small changes to largely already correct code.

One of the primary reasons that abstraction is overloved is that a completed program full of the right abstractions is perfectly beautiful. But there are very few completed programs, because programs are written, maintained, bugs are fixed, features are added, performance is tuned, and a whole variety of changes are made both by the original and new programming team members. Thus, the way a program looks in the end is not important because there is rarely an end, and if there is one it isn't planned.

What is important is that it be easy for programmers to come up to speed with the code, to be able to navigate through it effectively, to be able to understand what changes to make, and to be able to make them safely and correctly. If the beauty of the code gets in the way, the program is not well written, just as an office building designed to win design awards is not well designed when the building later must undergo changes but those changes are too hard to make. A language (and an accompanying environment) is poorly designed that doesn't recognize this fact, and worse-still are those languages that aim for the beauty and elegance of the (never) finished program.

Habitability is not clarity, I think. If it were, then Modula-3 would hit the nail on the head. Clarity is just too rare, and it's dangerous, too, in a funny way.

Books on writing tell you to be clear and simple, to use plain language. What can this mean? It is no advice at all because no one would ever try to be muddy and complex, to use obscure language as part of an *honest* attempt to be understood. But clarity is a quality that is rarely achieved—rare because it is difficult to achieve. However, even though only the rare writer or poet throws off an occasional clear sentence or paragraph, many of us are able to be understood with little problem, because our writing is habitable—not great, but easy enough to read, easy enough to change.

The danger of clarity is that it is uncompromised beauty; and it's real tough to to improve uncompromised beauty. Many second- and third-rate sculptors can fix a decent sculpture—I saw a group of them one summer making replacement gargoyles for Notre Dame Cathedral in Paris—but which of them would dare repair Michelangelo's *David*? Who would add a skyscraper to the background of *Mona Lisa*? Who would edit Eliot's poems? Clarity is dangerous.

If a programming language is optimized for the wrong thing—like pleasing efficiency, mathematical precision, or clarity—people might not be able to live with in or in it: It isn't habitable, piecemeal growth isn't possible, and the programmers who must live in the software feel no responsibility or ownership.

Abstraction Descant

The room fills with cold, conditioned air; outside the heat hazes, filtered through greened glass windows: a new building hardly first-populated. The speaker is wild-eyed, explaining new ideas like a Bible thumper. His hair is a flat-top; his mouth frowns in near grimace. He strides to my seat, looks down and says in a Texas drawl, "and the key is simply this: Abstractions. New and better abstractions. With them we can solve all our programming problems."
(Gabriel and Steele 1990)

This scene, which occurred in the late 1980s, began my off-and-on inquiry into the good and bad points of abstraction. All through my computer science education, abstraction was held up as the bright, shining monument of computer science. Yet when discussed in essays on the philosophy of science, abstraction is routinely questioned. Here is what Paul Feyerabend says about Ernst Mach, scientist and philosopher:

We have seen that abstraction, according to Mach, "plays an important role in the discovery of knowledge." Abstraction seems to be a negative procedure: real physical properties . . . are omitted. *Abstraction, as interpreted by Mach, is therefore* "a bold intellectual move." *It can misfire, it "is justified by* success." (Feyerabend 1987)

Throughout the book abstraction is discussed in terms of its relation to reality and to the "interests" of those who develop them. Here is what Bas Van Fraassen says:

But in certain cases, no abstraction is possible without losing the very thing we wish to study. . . .Thus [study at a certain level] can only go

> *so far—then it must give way to less thorough abstraction (that is, a less shallow level of analysis).* (Van Fraassen 1980)

Both Van Fraassen and Feyerabend seem to subscribe to the notion that abstraction in the realm of science—I doubt they would include computer science—is about ignorance. Abstraction ignores or omits certain things and operates at a shallow level of analysis.

The following is the definition of abstraction in computer science that I've used for years:

> Abstraction *in programming is the process of identifying* common patterns *that have* systematic variations*; an abstraction represents the common pattern and provides a means for specifying which variation to use.*
>
> *An abstraction facilitates separation of concerns: The implementor of an abstraction can ignore the exact uses or instances of the abstraction, and the user of the abstraction can forget the details of the implementation of the abstraction, so long as the implementation fulfills its intention or specification.* (Balzer et al. 1989)

This definition does not directly mention ignorance or omission, though it does imply them. The common pattern is omitted—in most types of abstraction the common pattern is replaced by a name, such as function name, a macro name, or a class name. The structure and behavior of the common pattern is lost except that the name of the abstraction denotes it and the definition of the abstraction (its implementation) contains it. And the foregoing definition does mention the interests of particular people—abstraction implementers and abstraction users.

Although I will return to the themes brought out by Feyerabend and Van Fraassen later, first I will set the scene. In the first two essays in this book I looked at three new concepts for programming: *compression*, *habitability*, and *piecemeal growth*, defined as follows:

> Compression *is the characteristic of a piece of text that the meaning of any part of it is "larger" than that particular piece has by itself. This characteristic is created by a rich context, with each part of the text drawing on that context—each word draws part of its meaning from its surroundings.*
>
> Habitability *is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions, and to change it comfortably and confidently.*

Piecemeal growth *is the process of design and implementation in which software is embellished, modified, reduced, enlarged, and improved through a process of repair rather than of replacement.*

Habitability can be limited by various things; abstraction being one of them. Many of my comments about abstraction also apply to encapsulation, which is a strong variety of abstraction. But don't panic: Abstraction and encapsulation are good things; my point is that they should be used only in moderation.

One bad effect of abstraction on programs occurs when it is taken too far. This results in code that is structured like a big pyramid, in which one level of abstractions is built tightly on another with minimal interfacial glue. Wait, that's not right—abstractionists would have you believe that abstraction guarantees you'll build geometrically precise programs, but that's garbage. Even in single inheritance object systems (in which hierarchies look like pyramids), there are other use relationships besides inheritance. Let's try this paragraph again.

One bad effect of abstraction on programs occurs when abstraction is taken too far. This results in code that is structured like a big web, in which groups of abstractions are tightly interwoven with others by means of interfacial glue.

In such large tangles lies one of the problems. If one abstraction is used in many places and that abstraction's interface is wrong, then repairing it forces repair of all its uses. The more the abstraction is shared, the more repair will be needed. But such repair is dangerous because it is usually made with less deliberation than the original design and in the context of unexpected requirements. If the repairs are being made by people other than the original designers and much later than the design and original implementation, the likelihood of mistake or ugliness will be increased.

Such webs are examples of compression: The meaning of an expression written in the context of the web is determined by the contents of the entire web. If you need to change another part of the web, your compressed expression might change its meaning, what it does, or even whether it works at all. So when we build our tight inheritance hierarchy in object-oriented fashion—weblike or pyramid style—we might be falling into this trap.

The reason for this failure is the insistence on using abstractions throughout, at all levels. If, instead, there were a level at which the code became direct rather than indirect, this might be less likely to happen. The problem is that people are taught to value abstraction above all else, and object-oriented languages and their philosophy of use emphasizes reuse (compression), which is generally good. However, sometimes the passion for abstraction is so strong that it is used inappropriately—it is forced in the same way as it is with larger, more complex, and typically ad hoc abstractions.

Abstractions must be carefully and expertly designed, especially when reuse or compression is intended. However, because abstractions are designed in a particular context and for a particular purpose, it is hard to design them while anticipating all purposes and forgetting all purposes, which is the hallmark of the well-designed abstractions.

This implies that abstractions are best designed by experts. Worse, average programmers are not well-equipped to design abstractions that have universal usage, even though the programming languages used by average programmers and the programming language texts and courses average programmers read and attend to learn their trade emphasize the importance of doing exactly that. Although the designers of the programming language and the authors of texts and course instructors can probably design abstractions well, the intended audience of the language—average programmers—cannot and are therefore left out. That is, languages that encourage abstraction lead to less habitable software, because its expected inhabitants—average programmers working on code years after the original designers have disappeared—are not easily able to grasp, modify, and grow the abstraction-laden code they must work on.

Not everyone is a poet, but most anybody can write usable documentation for small programs—we don't expect poets to do this work. Yet we seem to expect that the equivalent of poets will use high-level programming languages, because only program-poets are able to use them. In light of this observation, is it any wonder that abstraction-poor languages like C are by far the most popular and that abstraction-rich ones like Lisp and Smalltalk are niche languages?

Recall that one of the virtues of abstraction is that "the user of the abstraction can forget the details of the implementation of the abstraction." Nonetheless, the interface of a complex abstraction is likely to expose the implementation because portions of the abstraction's interface are likely to be dedicated to accessing particular behavior. Inexpertly designed and ad hoc abstractions frequently suffer from this shortcoming. Once the implementation is even partially revealed, it becomes more difficult to change the abstraction's implementation without causing problems. If abstraction were limited to simple cases, especially those with a fairly universal meaning—like the well-loved and ubiquitous stack, a favorite of almost every paper on abstract types—this problem would be reduced.

Another problem with complex abstraction arises from the observation that *abstractions are about ignorance*. The prime idea of encapsulation is that the implementation is hidden, thereby preventing assumptions about the implementation. Some complex abstractions, however, contain information about the implementation that is legitimately required, such as its performance, the algorithm, coding tricks, and resource usage—keep in mind that almost all interaction issues are about resource conflicts. When the implementation is truly hidden, its user is forced to use real scientific methods to infer or discover the needed information. Rather than preventing assumptions, hard encapsulation tends to guarantee incorrect assumptions.

Furthermore, when the implementation is truly hidden and there is a need to make a change or add behavior, the user is reduced to reinventing; if the implementation is not only hidden but also protected, the user will need to copy and maintain a parallel version. This implies that later changes to the code will be less efficiently made because similar code must be implemented in several places. In fact, the programmer—especially a programmer new to the project—may be unable to find all the occurrences of the similar code.

Notice the dilemma: Complex abstractions sometimes reveal implementation, which limits the opportunity to change their implementations, and they also are intended to hide implementation, which forces programmers (inhabitants) into ignorance and its resulting frustration, ineffectiveness, and feelings of denied responsibility. Strict abstractionists would argue that it is far better to hide the implementation, but consider what Christopher Alexander said:

> *When people lose the sense of responsibility for the environment they live in, and realize that they are merely cogs in someone else's machine, how can they feel any sense of identification with the community, or any sense of purpose there?* (Alexander 1975)

Why put your programmers through this?

Most of these problems can be eliminated or reduced if the user is encouraged to build small abstractions only. This is easier if the language provides other mechanisms that help the programmer build the larger structures needed in a program, structures that would normally be constructed from abstractions.

When we couple this advice with that of building hierarchies slowly, keeping them shallow as long as possible, we might find that we have paved the way for habitability, piecemeal growth, and healthy, usable compression. In fact, if we put in place an explicit process of piecemeal growth, hierarchies will naturally grow slowly, and they are more likely to be correct, because their use evolves over time. The resulting compression also is natural, not forced, and it is less likely that the compression will backfire.

This same advice is more cynically arrived at by Martin D. Carroll and John F. Isner when discussing the design of the C++ Standard Components developed by UNIX Systems Laboratories:

> *[W]e take the minimalist approach to inheritance. We use it only when it makes our components more efficient, or when it solves certain problems in the type system.*
>
> *We do not intend for our components to serve as a collection of base classes that users extend via derivation. It is exceedingly difficult to make a class extensible* in abstracto *(it is tenfold harder when one is trying to provide classes that are as efficient as possible). Contrary to a common misconception, it is rarely possible for a programmer to*

> *take an arbitrary class and derive a new, useful, and correct subtype from it, unless that subtype is of a very specific kind anticipated by the designer of the base class.* Classes can only be made extensible in certain directions, where each of these directions is consciously chosen (and programmed in) by the designer of the class. *Class libraries which claim to be "fully extensible" are making an extravagant claim which frequently does not hold up in practice. . . . There is absolutely no reason to sacrifice efficiency for an elusive kind of "extensibility."* (Carroll and Isner 1992, emphasis added)

Their argument comes more from efficiency than from the needs of habitability and piecemeal growth; in fact, their whole essay is about how to achieve what they consider the most important characteristic of a library—its efficiency. But notice that they make a very strong claim that being able to extend a class hierarchy is nearly impossible, though they don't say why precisely. One can try to puzzle out the reason; the best I can come up with is that many possible extensions require access to the internal implementation of a class, and the author of the class either does not know what parts are needed or wishes to retain the opportunity to change the implementation and therefore hides the implementation. In C++ there are other reasons, like the need for speed encourages class authors to make virtual as little as possible.

Regardless of the reasons behind it, if we accept Carroll and Isner's statement that unanticipated extension is difficult, their statement supports my claim that during the growth of an abstraction web, one must frequently go back to existing abstractions to repair them, such repair being a sort of delayed anticipation of the extensions.

Because abstracting a common pattern and using the abstraction is a process of replacing something directly expressed by a shorthand, adding an abstraction is like adding a new word to a real language: It requires learning a new vocabulary. Real languages rarely permit us to add new words, and such words are reserved for concepts, objects, actions, and qualifications that are basic or newly basic. We don't invent new words for dogs that can jump through hoops or automobiles modified to carry 300 gallons of fuel. First, there is no need for these words except within a small familylike circle, and even there the need soon dissipates when the name of the dog or a phrase like "Joe's big fueler" prove worthy and effective.

Second, these words would be meaningless outside the small circle that might need them, and encountering an enclave of odd-language users would negate the advantages of natural language. Similarly, we want to limit the use of abstraction to either common items—common to every programmer—or items inextricably linked to the application domain. Otherwise, unnecessarily abstracted code would be unintelligible to programmers fresh to a project, and hence it would not be habitable.

Van Fraassen hints at this idea: "But in certain cases, no abstraction is possible without losing the very thing we wish to study." He means that the object of interest is captured only by the name of the abstraction rather than by the thing itself. For example, when we speak of redness and *go no deeper*, redness disappears and all we have left is the name *red*.

Recall that one of the primary reasons that abstraction is overloved is that a completed program full of the right abstractions is perfectly beautiful—it is "justified by success" as Mach said. Of course, Christopher Alexander would probably not think such a program was beautiful—assuming he would even know how to read a program. Note what he said about completed, planned urban development:

> *[P]lanned development is also generally not coherent . . . not in a deep-felt sense. It is supposed to be. But if we ask ourselves whether the final product of current urban design projects actually is coherent in the real, deep sense that we know from traditional towns, then we must say no. The order is superficial, skin deep, only in the plan or in some contrived orderliness of the arrangements. There is no deep inner coherence, which can be felt in every doorway, every step, and every street.*
>
> *And . . . this modern planned development which we think of as normal, certainly has NO power to evoke deep feeling. It can, at best, ask for some kind of admiration for "design." But of deep feeling there is no word, not a tremor, not a possibility.* (Alexander 1987)

Alexander's area of study—architecture and urban development—has the advantage of having had a long history and, hence, examples of the sort of habitable, deeply felt homes and towns and cities with which he can contrast the products of the modern design approach. In software there are no large examples that I can point to with which we are all familiar. And so we can ask whether Alexander's words—which sound so nice when apply them to the field of software—pertain in any real sense to computer science. But that is a question for another essay.

Remember, piecemeal growth is the norm, and programs grow, change, and are repaired. Therefore, the perfectly beautiful program is possible only for very small programs, like the ones in books on programming or in programming courses.

Now let's return to my definition of abstraction. One thing that strikes me about this definition, now that I have used it for years and years, is the degree to which the fact is ignored or forgotten that there are two conditions for abstraction. There is not only a common pattern but also systematic variations. Sub–routines and functions are perfect examples: The common pattern is the code that ends up in the subroutine, and the systematic variations are the arguments that are passed—the mechanism for specifying the variation is the parameter.

What about common patterns without systematic variations? I will turn to this topic in the second part of this essay.

To some extent, object-oriented programming is about piecemeal growth. But in some languages, it is achieved by catching the excess-abstraction disease. This essay is not intended to convince you that abstraction is bad (or that I'm nuts), but to show that maybe some of the other concerns of software development—creating habitable software that can be effectively maintained, recognizing that the reality of software development is piecemeal growth and to plan accordingly, and to understand that the power of object-oriented programming is compression, which carries a terrific requirement for careful use of inheritance—relate to how we use abstraction and how much we use it.

In order for the building to be alive, its construction details must be unique and fitted to their individual circumstances as carefully as the larger parts. . . . The details of a building cannot be made alive when they are made from modular parts. (Alexander 1979)

Let's look again at the definition of abstraction I've been using:

> Abstraction *in programming is the process of identifying* common patterns *that have* systematic variations*; an abstraction represents the common pattern and provides a means for specifying which variation to use.*
>
> *An abstraction facilitates separation of concerns: The implementor of an abstraction can ignore the exact uses or instances of the abstraction, and the user of the abstraction can forget the details of the implementation of the abstraction, so long as the implementation fulfills its intention or specification.* (Balzer et al. 1989)

Unfortunately, such patterns are usually turned into abstractions, with the unsystematic variations being lumped in with the systematic ones, the result being an ad hoc interface. Using such ad hoc abstractions, client code—code using an abstraction—can end up being composed of glue code surrounding "invocations" of the ad hoc abstractions. The glue code, however, may only to contort the natural structure of the client code. Notice how this effect compounds another problem with abstractions: If an abstraction is composed of what amounts to a conglomeration of somewhat related operations, programmers are more likely to want to modify or add to the abstraction—it's ugly enough to begin with, so what's the harm?

Examples of this sort of abuse of abstraction abound in Common Lisp—though I like the language and don't intend to criticize it. A typical example is `mismatch`, which searches two sequences (a sequence is a Common Lisp data type) for the first index where the two sequences differ according to a test predicate. Common Lisp supports *keyword arguments*, which provide a means to supply a variable number of optional arguments by naming them. This mechanism helps clarify the meaning of (the possibly many) arguments to a complex function.The function `mismatch` takes a variety of optional arguments that specify the direction to search, a function to test whether two items are the same, and whether the predicate should be negated (even though there are mechanisms to negate predicates in the language and the negated predicate can be easily passed to `mismatch`). Clearly, Common Lisp, though a well-designed language, contains a number of common patterns without systematic variations.

Christopher Alexander's comment on modular parts, which began this essay, bears on this point. Contrast it with what Greg Nelson said about Modula-3:

> *The better we understand our programs, the bigger the building blocks we use to structure them. After the instruction came the statement, after the statement came the procedure, after the procedure came the interface. The next step seems to be the* abstract type.
> (Nelson 1991)

What Alexander seems to be saying is that if we try to use modular parts (solid building blocks not easily molded to the current circumstances), then the overall structure of the thing built with them may become overly constrained by their shape. In programming, if a set of large abstractions does nearly the right thing, it is tempting to use them and to bend the structure of the surrounding program to fit them. This can lead to uninhabitable programs.

Worse: You can fight this temptation and choose *not* to use them. This choice also can lead to uninhabitable programs because you will be using parts similar but subtly different from possibly familiar ones. The only way to avoid this is to use small blocks rather than large ones, or to use blocks well-designed and tested by experts.

Large abstractions are large common patterns, and what is missing in programming languages is a treatment of common patterns. Such a treatment would need to support separating their use from their definition. Now, what would such a separation mean in ordinary abstractions? The key benefit is that there would be just one place the programmer has to look in order to repair or study it. There is no reason a language or—far better—a programming environment couldn't show the underlying common parts of a pattern. Later I'll cite an example of what this could be like.

What is an example of such a pattern? The idea of accumulating a result is one:

```
(let ((result ...)
       ...)
  ...
  (<loop> ... (setq result ...) ...)
  ...result...)
```

This isn't such an interesting example because the pattern is so familiar, but it is easy to see it as a pattern people learn and which cannot be easily captured by a traditional abstraction. There isn't too much one can do to systematically modify this pattern—it has so few common parts and so many potential variations—but there is a lot that programmers gain from knowing this pattern and later coming upon a piece of code that contained it: It would help them know why the obscure-looking variable is popping up from place to place; it would help them see in their mind's eye the whole pattern with the intervening portions elided.

One way to lay a foundation of common patterns is the same way we do with natural language: Teach people the most common patterns. We never think to teach people how to create words—poets do this frequently, and sometimes ordinary people become word-inventors—yet we teach budding programmers to create their own vocabulary but we don't provide a catalog of common patterns of usage.

I think this means that we need to spend more time teaching programming, and the increased time should be devoted to teaching patterns and reading "great" programs. How much time do we spend reading in our ordinary education? And from our reading we gain a foundation for writing. In turn, this foundation is sometimes expanded by careful instruction and tutoring in writing. Certainly many people who write for a living go through this process. But in programming we just learn the language and solve a bunch of short puzzles. Sort of like writing 50 limericks and then off to write books.

Let's look at a simple example of how abstractions and patterns interact. Consider the following code fragment:

```
(let ((result (mapcar f list))) ...)
```

This takes a function `f`, applies it to each element of the list `list`, and binds the list of the results of those applications to the variable `result`. Now, an ordinary programmer knows that `mapcar` traverses the list, so there should be some tricky way to make the `mapcar` do double duty, computing the length of the list as well. But try as you might, you won't find an easy way to modify `f` so that the length can be transparently obtained and abstraction boundaries heartily enforced—remember I said an easy way.

There will always be some object somewhere that is accumulating the length; we can bury the side-effecting code in a modified `f`, but there still is the access of that accumulator, whether it be a variable or some object. If you really wanted to have abstraction reign, you'd try to write something like this:

```
(let  ((result  (mapcar f list) )
        (len    (length  list)  )) ... )
```

Here the shaded area indicates a place where a "sufficiently smart" compiler could try to fold the two computations into one. Actually, there are lots of ways to imagine an environment helping promote such transparent patterns. One is that this could be the environment's surface manifestation of an optimized section of code. In this case, the programmer might have made the optimization by hand, and the environment would be simply showing the unoptimized code; this environmental feature might also aid with piecemeal growth by maintaining a history of the software evolution.

This formulation also correctly respects the abstraction of both operations—the `mapcar` and the length computation—but it incorrectly overemphasizes abstraction because the unabstracted but common-pattern code is just fine:

```
(let ((length 0))
  (let ((result
          (mapcar
            #'(lambda (x)
                (incf length)
                (f x))
            list)))
    ...))
```

This code is obvious to even a novice (modern) Lisp programmer, and it requires a lot less mechanics than does the environmental approach just before it. Further, note that the length computation winds like a vine from the outside of the `mapcar` abstraction to its inside.

Let's look at another problem with abstractions: Data and control abstractions are generally best when they are codesigned and this is rarely done anymore. Consider, for example, the FORTRAN abstractions of arrays and iteration. Arrays are abstractions designed to represent vectors and matrices. Iteration is a control abstraction useful for traversing vectors and arrays. Think, for example, of how easy it is to implement summation over the elements of a vector. This is because arrays and `DO` loops were codesigned.

The codesign of mathematical data and control abstractions is not an accident. One could hardly ignore the need to refer to individual elements in a matrix or a sequence while performing nested sums and products in a numeric computation.

Partly the codesign was made by mathematicians before Fortran was created, and partly because when Fortran was developed there were no alternatives other than conditional and `GO` statements. The success of Fortran is due at least somewhat to the close match between the data abstractions (scalars, vectors, and arrays) and control abstractions that manipulate them (`DO` loops).

Even though there was a recognized need to be able to define data abstractions after Fortran was developed, there was never a recognized need to be able to define control abstractions. Some languages, like Lisp, adopted a powerful macro facility which enables programmers to define their own control abstractions. Of course, macros also enable programmers to define their own data structures by providing a means to define a protocol that is syntactically the same as ordinary function invocation.

But an interesting thing happened to Lisp in the early 1980s: the use of macros to define control structures became forbidden style. Not only did some organizations outlaw such use of macros, but the cognoscenti began sneering at programmers who used them that way. Procedural abstractions are acceptable, but not control abstractions. The only acceptable control abstractions in Lisp today are function invocation, `do` loops, `while` loops, `go` statements (sort of), non-local exits, and a few mapping operations (such as `mapcar` in Lisp).

The `mismatch` example shows how one sort of abstraction (a function) can be used to implement control abstractions. Some of the arguments specify the function's control behavior (which direction to search, how to extract the data of interest, how to test whether the item was found, and whether to negate the value of that test function). The common pattern—generalized search, generalized extraction, generalized test, and gratuitous negation—has been completely eliminated, and all hope of understanding a code fragment invoking this abstraction rests with being able to understand the name of the function and the meanings of its arguments. Common Lisp, at least, provides keyword arguments to name the role of the arguments. Does the following code fragment:

```
(mismatch sequence list :from-end t
          :start1 20 :start2 40
          :end1 120 :end2 140 :test #'baz)
```

seem easier to understand than this pattern of use:

```
(let ((subseq1 (reverse (subseq sequence 20 120)))
      (subseq2 (reverse (subseq list 40 140))))
  (flet ((the-same (x y) (baz x y)))
    (loop for index upfrom 0
          as item1 in subseq1
          as item2 in subseq2
          finally (return t) do
      (unless (the-same item1 item2)
        (return index)))))
```

This latter code fragment is an example of a common pattern. If you have been taught to see such patterns, they are as easily understood as the shorthand `mismatch` call. Furthermore, if you have not been trained to understand either the `mismatch` or the common pattern, you can still understand the common pattern just by reading it. The `mismatch` expression has two advantages over the common pattern one:

- System implementers can more easily guarantee that the implementation of `mismatch` is maximally efficient, coding it in assembler if need be (I'm sure you'll rest better tonight knowing that).
- It is harder to type the longhand common pattern than the `mismatch` expression, and it is just plain longer.

The first problem goes away when computers are fast enough (more on this later). The second goes away with a well-designed programming environment or doesn't matter at all when you consider the habitability gains from using an easily understood program. In fact, there is no reason that the shorthand `mismatch` expression could not be simply an abbreviation for the longhand common pattern. The programmer could decide which to see and could change either one. Then if the common pattern version strays too far, it will no longer be abbreviated (because it can't be).

Let's consider the scenario in which the interface to a data abstraction is being extended—we've added to the interface and we need to modify our program to invoke the new parts. For example, additional state must be initialized and maintained. The problem of adding invocations of the new part of the protocol is exacerbated by the fact that only mathematical, FORTRAN-like control structures are available. Over the years programmers have gotten into the habit of optimizing the use of these control structures for either efficiency or style. Typically the most compact control structure for the specific job at hand is often preferred to a more general formulation. Modifying such optimized control structures sometimes requires large a structural modification of a program when only small modifications seem necessary. For instance, the initialization information might not be available at the point we need it and must be reconstructed or saved somewhere, or only part of the initialization information might have been computed because, earlier, not all of it was needed.

This problem with abstractions stems from mixing a fixed set of (inappropriate) control abstractions with custom-designed data abstractions. If natural control abstractions were matched, they must be implemented nonabstractly using existing low-level primitives—that is, through other control structures. Even though there is a pattern of control, there is no way to abbreviate it except through procedural or functional abstractions.

Such lopsided use of data abstraction forces programs to be written at varying levels of abstraction. The result is that the programmer is reduced to switching mentally between these levels. Right next to a protocol invocation that represents the opening of a floodgate will be the assignment of 1 to a flag that tells a later part of the program that a certain initialization already took place. Furthermore and worse, the flag might be part of an otherwise completely application domain-level data abstraction.

The use of procedural or functional abstractions (combined with the fact that argument evaluation rules might thwart a need to pass expressions and not values) only pushes the problem down one level: Within the procedure or function the implementation of the control abstraction is fully exposed, even though objects being manipulated are high-level data abstractions.

Regardless of what you make of this view of data versus control abstraction, it is certainly true that because almost every programming language does not allow any sort of meaningful user-defined control abstractions, there is always a mismatch in abstraction levels between control and data. If there is a good reason for allowing data abstractions, why isn't that a good reason for allowing control abstractions; and if there is a good reason to disallow control abstractions, why isn't that a good reason to disallow data abstractions? Nevertheless, it is accepted practice to use existing control abstractions to implement others using common patterns. My argument is that perhaps we should be more willing to use common patterns for other things as well.

The real reason that common patterns are not used rather than tight abstractions is efficiency. It is more efficient to write abstracted, compressed code than uncompressed common patterns, and it is more efficient to execute abstracted code in some cases. For example, if we were to write the two lines that do `mapcar` and `length`, they would run about twice as slow as some complex compressed version. This wouldn't matter if computers were big enough and fast enough for the programs we need, but right now they aren't. So we continue to pay with the sweat of people so that computers can take it easy and users don't have to be inconvenienced. Perhaps someday the economics of this situation will change. Maybe not.

Common patterns are similar in nature though not detail to the patterns that Christopher Alexander uses in his so-called pattern languages. A pattern language is a language for generating buildings and towns with organic order. Patterns generally specify the components of a portion of a building or a place and how those components are related to other patterns. Here is an example of a pattern that every planner of developers' offices should know:

> *Locate each room so that it has outdoor space on at least two sides, and then place windows in these outdoor walls so that natural light falls into every room from more than one direction.* (Alexander 1977a)

The bulk of Alexander's written work over the last 15 years—and from which I have been freely quoting—describes the theory behind pattern languages and the patterns for building and urban development that he and his students and colleagues have devised. These patterns and the social process for applying them are designed to produce organic order through piecemeal growth. Clearly there is a connection between patterns as Alexander defines them and the common patterns that form half the definition of abstraction. But there is no room in this essay to explore it; perhaps in another.

Here are the lessons that I think the last two essays plus this one teach:

- Object-oriented languages gain their power from compression, which can lead to compact, efficiently expressed programs. Compression, though, can present problems if it is prematurely used.
- Software development is always through piecemeal growth and rarely through thorough design. Such planned development can lead both to technical problems because the future of a piece of software cannot be known and also to social problems because completely planned development alienates those developers who are not also the planners.
- Habitability is one of the most important characteristics of software. It enables developers to live comfortably in and repair or modify code and design.
- When taken to extremes, abstraction can diminish habitability and can result in premature compression. Beware of overabstracting or of abstracting when a common pattern will do.
- There is much to learn about software development, and we are just starting to do that.

The Quality Without a Name

In 1992 I started reading the more recent work of Christopher Alexander, the Berkeley architect who studies design. The work I'm referring to is captured in the books, *The Timeless Way of Building* (1979), *A Pattern Language* (1977b), *The Oregon Experiment* (1975), and *A New Theory of Urban Design* (1987). Computer scientists over the years have picked up on his writing, and now a small group of them are into "writing patterns." Patterns certainly have an appeal to people who wish to design and construct systems because they are a means to capture common sense and are a way to capture abstractions that are not easily captured otherwise.

My own trek into the space of Alexander's thought began slowly—I read the work, but I tried not to jump to conclusions about its relation to software design. I wanted to figure out what the corresponding points were between architecture and software. The first place where I think I differed with others' interpretation of Alexander's work was in defining the users or inhabitants of a piece of software as its coders and maintainers. At least one computer scientist identified the "user" of a piece of software as the end user. This appears to make sense at first, but when you read Alexander, it is clear that a "user" is an inhabitant—someone who lives in the thing constructed. The thing constructed is under constant repair by its inhabitants, and end users of software do not constantly repair the software, though some might want to.

In earlier essays I've hinted that my trek might also head in the direction of patterns, which are not quite abstractions, modules, or classes. Alexander himself proposes pattern languages as a way to approach design at all levels, from cities and towns to houses and rooms and even to construction techniques.

Now I am at the point of trying to figure out what corresponds to Alexander's patterns. To do this, though, requires figuring out as precisely as I can what the *quality without a name* is in the realm of software. This quality is at the heart of everything Alexander has done since the mid-1960s, and it figures heavily in his conception of pattern languages. Pattern languages are designed to generate towns, communities, neighborhoods, buildings, homes, and gardens with this quality. Alexander's search, culminating in pattern languages, was to find an objective (rather than a subjective) meaning for beauty, for the aliveness that certain buildings, places, and human activities have. The objective meaning is the quality without a name, and I believe we cannot come to grips with Alexander in the software community unless we come to grips with this concept.

In an interview with Stephen Grabow, Alexander stated:

> *I was no longer willing to start looking at any pattern unless it presented itself to me as having the capacity to connect up with some part of this quality [the quality without a name]. Unless a particular pattern actually was capable of generating the kind of life and spirit that we are now discussing, and that [sic] it had this quality itself, my tendency was to dismiss it, even though we explored many, many patterns.* (Grabow 1983)

Computer scientists who try to write patterns without understanding this quality are quite likely not following Alexander's program, and perhaps they are not helping themselves and others as much as they believe. Or perhaps they are doing harm. So what is this quality without a name?

The quality is an objective quality that things like buildings and places can possess that makes them good places or beautiful places. Buildings and towns with this quality are habitable and alive. The key point to this—and the point that really sets Alexander apart from his contemporaries and stirs philosophical debate—is that the quality is objective. First I'll try to explain the quality, then I'll explain what is so radical about the concept of such a quality. Here is what Alexander says:

> *The first place I think of when I try to tell someone about this quality is a corner of an English country garden where a peach tree grows against a wall.*
>
> *The wall runs east to west; the peach tree grows flat against the southern side. The sun shines on the tree and, as it warms the bricks behind the tree, the warm bricks themselves warm the peaches on the tree. It has a slightly dozy quality. The tree, carefully tied to grow flat against the wall; warming the bricks; the peaches growing in the sun; the wild grass growing around the roots of the tree, in the angle where the earth and roots and wall all meet.*
>
> *This quality is the most fundamental quality there is in anything.* (Alexander 1979)

At first the quality sounds like one reserved for art or architecture. But Alexander asserts that the patterns themselves in his pattern languages must have the quality, and it's fairly clear from what he says about the quality that almost anything can have it or not.

Let's try to figure it out. Alexander says:

> *It is a subtle kind of freedom from inner contradictions.*
> (Alexander 1979)

This statement reflects the origins of his inquiry into the quality. It started in 1964 when he was doing a study for the Bay Area Rapid Transit (BART) system based on the work reported in *Notes on the Synthesis of Form* (Alexander 1964), which in turn was based on his Ph.D. dissertation. One of the key ideas in this book was that in a good design there must be an underlying correspondence between the structure of the problem and the structure of the solution— good design proceeds by writing down the requirements, analyzing their interactions on the basis of potential misfits, producing a hierarchical decomposition of the parts, and piecing together a structure whose

> *structural hierarchy is the exact counterpart of the functional hierarchy established during the analysis of the program.* (Alexander 1964)

Alexander was studying the system of forces surrounding a ticket booth, and he and his group had written down 390 requirements for what ought to be happening near it. Some of them pertained to such things as being there to get tickets, being able to get change, being able to move past people waiting in line to get tickets, and not having to wait too long for tickets. What he noticed, though, was that certain parts of the system were not subject to these requirements and that the system itself could become bogged down because these other forces—forces not subject to control by requirements—acted to come to their own balance within the system. For example, if one person stopped and another also stopped to talk with the first, congestion could build up that would defeat the mechanisms designed to keep traffic flow smooth. Of course there was a requirement that there not be congestion, but there was nothing the designers could do to prevent this by means of a designed mechanism.

Alexander said this:

> *So it became clear that the free functioning of the system did not purely depend on meeting a set of requirements. It had to do, rather, with the system coming to terms with itself and being in balance with the forces that were generated* internal *to the system, not in accordance with some arbitrary set of requirements we stated. I was very puzzled by this because the general prevailing idea at the time [in*

> *1964] was that essentially everything was based on goals. My whole analysis of requirements was certainly quite congruent with the operations research point of view that goals had to be stated and so on. What bothered me was that the correct analysis of the ticket booth could not be based purely on one's goals, that there were realities emerging from the center of the system itself and that whether you succeeded or not had to do with whether you created a configuration that was stable with respect to these realities.* (Grabow 1983)

A system has this quality when it is at peace with itself, when it has no internal contradictions, when it is not divided against itself, when it is true to its own inner forces. And these forces are separate from the requirements of the system as a whole. In software we hear about gathering requirements, through talking to users or customers or by examining the problem space. In the world of computer-aided software engineering (CASE) we hear about traceability, which means that every procedure or object can be traced back to the requirement that spawned it. But if Alexander is right, then many of the key characteristics of a system come from internal forces and not external requirements. So to what will such parts of system trace? Perhaps to the requirement that a system have the quality without a name.

Alexander proposes some words to describe the quality without a name, but even though he feels they point the reader in a direction that helps comprehension, these words ultimately confuse. The words are *alive*, *whole*, *comfortable*, *free*, *exact*, *egoless*, and *eternal*. I'll go through all of them to try to explain the quality without a name.

The word *alive* captures some of the meaning when you think about a fire that is alive. Such a fire is not just a pile of burning logs, but a structure of logs in which there are sufficient and well-placed air chimneys within that structure. When someone has built such a fire, you don't see them push the logs about with a poker but you do see them lift a particular log and move it an inch or maybe a half inch, so that the air flows more smoothly or the flame curls around the log in a specific way to catch a higher-up log. Such a fire burns down to a small quantity of ash. This fire has the quality without a name.

The problem with the word *alive* is that it is a metaphor—it is hard to know whether something literally not alive, like a fire, is, in fact, *alive*, and when we try to think of what makes a fire alive, we're not really sure.

Whole captures part of the meaning, because for Alexander a thing that is whole is free from internal contradictions or inner forces that can tear it apart. The analogy he uses is a ring of trees around the edge of a windblown lake: The trees bend in a strong wind, and the roots of the trees keep the bank from eroding, and the water in the lake helps nourish the trees. Every part of the system is in harmony with every other part. On the other hand, a steep bank with no trees is easily eroded—the system is not whole, and the system can destroy itself: the grasses and trees are destroyed by the erosion, the bank is torn down, and the lake is filled with mud and disappears. The first system of trees, bank, and lake has the quality without a name.

The problem with this word is that *whole* implies, to some, being enclosed or separate. A lung is whole but it is not whole while still completely contained within a person—a lung requires air to breathe, which requires plants to absorb carbon dioxide and to produce oxygen. The system is much larger than the one that contains the lungs.

The word *comfortable* involves more that meets the eye. Alexander explains it this way:

> *Imagine yourself on a winter afternoon with a pot of tea, a book, a reading light, and two or three huge pillows to lean back against. Now, make yourself comfortable. Not in some way you can show to other people and say how much you like it. I mean so that you really like it for yourself.*
>
> *You put the tea where you can reach it; but in a place where you can't possibly knock it over. You pull the light down to shine on the book, but not too brightly, and so that you can't see the naked bulb. You put the cushions behind you and place them, carefully, one by one, just where you want them, to support your back, your neck, your arm: so that you are supported just comfortably, just as you want to sip your tea, and read, and dream.*
>
> *When you take the trouble to do all that, and you do it carefully, with much attention, then it may begin to have the quality with no name.* (Alexander 1979)

The problem with *comfortable* is that it has too many other meanings. For example, a family with too much money and a house that is too warm also is comfortable.

The word *free* helps define the quality by implying that things that are not completely perfect or overplanned or precise can have the quality too. It also frees us from the confines and limitations of *whole* and *comfortable*.

Free, of course, is not correct because it can imply reckless abandon or not having roots in its own nature.

The word *exact* counterbalances *comfortable* and *free*, which can give the impression of fuzziness or overlooseness. The quality *is* loose and fluid, but it involves precise, exact forces acting in balance. For example, if you try to build a small table on which to put birdseed in the winter for blackbirds, you must know the exact forces that determine the blackbirds' behavior so that they will be able to use the table as you planned. The table cannot be too low because blackbirds don't like to swoop down near the ground, and it cannot be too high because the wind might blow them off course, it cannot be too near to things that could frighten the birds like clotheslines, and it cannot be too exposed to predators. Almost every size for the table and every place to put it you can think of won't work. When it does work, the birdseed table has the quality with no name.

Exact fails because it means the wrong sort of thing to many people. Alexander says:

> *Usually when we say something is exact, we mean that it fits some abstract image exactly. If I cut a square of cardboard and make it perfectly exact, it means that I have made the cardboard perfectly square: its sides are exactly equal: and its angles are exactly ninety degrees. I have matched the image perfectly.*
>
> *The meaning of the work "exact" which I use here is almost the opposite. A thing which has the quality without a name never fits any image exactly. What is exact is its adaptation to the forces which are in it.* (Alexander 1979)

Egoless conveys an important and surprising aspect of the quality. I'll let Alexander say it:

> *When a place is lifeless or unreal, there is almost always a mastermind behind it. It is so filled with the will of the maker that there is no room for its own nature.*
>
> *Think, by contrast, of the decoration on an old bench—small hearts carved in it; simple holes cut out while it was being put together—these can be egoless.*
>
> *They are not carved according to some plan. They are carefree, carved into it wherever there seems to be a gap.* (Alexander 1979)

The word *egoless* is wrong because it is possible to build something with the quality without a name while retaining some of the personality of its builder.

Finally is the word *eternal*. By this word Alexander means that something with the quality is so strong, so balanced, so clearly self-maintaining that it reaches into the realm of eternal truth, even if it lasts for only an instant.

But *eternal* hints at the mysterious, and there is nothing mysterious about the quality. Alexander concludes his discussion of this quality with the following:

> *The quality which has no name includes these simpler sweeter qualities. But it is so ordinary as well that it somehow reminds us of the passing of our life.*
>
> *It is a slightly bitter quality.* (Alexander 1979)

This slightly bitter quality is at the center of Alexander's pattern languages. I believe that if we are to embrace pattern languages, we must also embrace this quality. But what is this quality in software? Certainly I am bitter when I think about some software I know of, but this isn't what Alexander is after. I'll return to this after explaining why this quality is regarded as revolutionary.

What is revolutionary about Alexander is that he is resuming the quest for an understanding of objective quality that science and philosophy abandoned in the modern era. In the seventeenth and eighteenth centuries, a tension developed in which mind and matter were separated by science and philosophy. From this came the separation of *fact* and *value*. After the separation, a fact had no value associated with it, a fact could not be good or bad, it just was. Science, then, tried to find theories that explained things as they were and no longer sought what was good or beautiful about things. That is, we no longer sought the objective characteristics of beauty, which is where Alexander started his quest.

Today it is hard for us to understand that fact and value once were tied together, and it is hard for us as software designers to think of what there could be about a software system that would exhibit Alexander's quality without a name. And, even if we could, it would be difficult to not dismiss it as something only in the eye of the beholder.

The study of beauty stopped because beauty became a mere contingency—whether something was beautiful didn't depend much or at all on the thing, only on the thing as perceived by an unnecessary observer. A thing was beautiful *to* someone: It was not simply beautiful.

Alexander stepped forward and tried to reverse the separation of fact from value. His program was not only to find patterns that explain the existence of the quality without a name but also to find patterns that generate objects with that quality. Furthermore, the patterns themselves must demonstrate the same quality.

Here is how Alexander puts it:

> *Myself, as some of you know, originally a mathematician, I spent several years, in the early sixties, trying to define a view of design, allied with science, in which values were also let in by the back door. I too played with operations research, linear programming, all the fascinating toys, which mathematics and science have to offer us, and tried to see how these things can give us a new view of design, what to design, and how to design.*
>
> *Finally, however, I recognized that this view is essentially not productive, and that for mathematical and scientific reasons, if you like, it was essential to find a theory in which value and fact are one, in which we recognize that here is a central value, approachable through feeling, and approachable by loss of self, which is deeply connected to facts, and forms a single indivisible world picture,* within which productive results can be obtained. (Alexander 1977a, emphasis in original)

For many, Alexander is merely pining for the days when quaint villages and eccentric buildings were the norm. But face it, the buildings he hates, you hate too; and buildings he loves, you love. If this is true, then maybe there is an objective value that we all can recognize.

Alexander is lucky that architecture has a very long history and that the artifacts of architecture from a lot of that history are visible today. This means that he can examine things built before science and philosophy relegated beauty to contingency. We in software are not so lucky—all of our artifacts were conceived and constructed firmly in the system of fact separated from value. But there are programs we can look at and about which we say, "no way I'm maintaining that kluge." And there are other programs about which we can say, "Wow, who wrote this!" So the quality without a name for software must exist.

One of the aspects of the quality with no name that Alexander seems clear on is that buildings with the quality are not made of large modular units. Some of his examples of buildings or things with the quality are built with bricks and other such small modular units, but not large ones. If you were thinking that Alexander is just talking about plain old highly abstract and modular code, perhaps you should think again. His pattern languages provide a well-thought-out abstract language for talking about buildings, and he encourages new pattern languages, but only when they embody the quality with no name—he does not endorse willy nilly pattern languages. Here is what he says:

> *Suppose, for example, that an architect makes the statement that buildings have to be made of modular units. This statement is already useless to me because I know that quite a few things are not made of modular units, namely people, trees, and stars, and so therefore the statement is completely uninteresting—aside from the tremendous inadequacies revealed by a critical analysis on its own terms. But even before you get to those inadequacies, my hackles are already up because this statement cannot possibly apply to everything there is in the universe and therefore we are in the wrong ballgame. . . . In other words, I actually do not accept buildings as a special class of things unto themselves, although of course I take them very seriously as a special species of forms. But beyond that is my desire to see them belong with people, trees, and stars as part of the universe.* (Grabow 1983)

If we look carefully at the buildings, towns, and cities that Alexander admires, we will see that they are European or perhaps even Third World. This suggests a couple of lines of inquiry into what the quality might be.

European building has an interesting constraint: There isn't much space. People and their buildings need to take up as little space as possible, and buildings, town layout, and common areas are cleverly put together to conserve what is precious. The same is true in many older cultures. Working under space constraints has a few interesting effects.

One effect is that things are on a smaller scale, where perfection is not as easily achieved and where regularity from irregular small parts is possible. Errors are seen as errors only in a context in which those errors are relatively large. When they are small, they form part of the attractiveness of approximate placement. One way to think about this is that nature has a regularity that is not captured well by precise prefractal geometry—such geometry is too precise, too exact to capture nature well. With fractals, though, we can simulate natural surroundings, and the approximate nature of small irregularities is mimicked by irregular placement and small errors.

Another effect is that of the creativity spawned by constraint, which is most easily explained by analogy to poetry. There are many reasons why poetry uses forms—specific meter, rhyme, and stanzaic structure. One reason is that such devices aid memory, and poetry was originally an oral art form. Another important reason is that by placing constraints on the form of a written statement, it becomes less easy to use a phrase or word that easily comes to mind, because it probably won't fit. And if it comes to mind easily, it is most likely a hackneyed phrase or at least one that the poet uses or hears frequently enough that it has lost its edge. By forcing the poet to write in a form, the easy, automatic phrase is eliminated, and the poet is forced to find other ways of saying what is meant. And when that isn't possible, the poet must look for other things to say. In both cases, the constraints force the poet to look for something new to say or new way to say it. Therefore, form in poetry is a device that helps create a climate or context for creativity and newness.

The same is true for construction: When there is no room to do the obvious or when a building must fit in a specific place, the architect must look for new solutions. Because constraint by form limits options, ways must be found to use a single space for multiple purposes.

Europe has a long history, and there are two more effects of that. One is that when people are put into a small space, building with flammable materials is dangerous. Therefore one must build with more durable and difficult materials. This implies that the standard of perfection must drop, and the results are buildings that look more like nature—more fractal-like.

The last effect is that today we see those buildings and towns that have survived because they are pleasant—there is a natural selection. It would be odd to see towns and buildings in Europe that are old and just plain ugly; they would not have survived.

Some of these things might be reasons to question whether the quality without a name really exists separate from the quality of looking like nature or the quality of being highly compressed.

Software has a situation corresponding to compression: bummed code. Bummed code is code that must perform a particular task in a highly constrained footprint because there just isn't any space to do the task in a straightforward manner. This often requires very clever encoding and multiple uses of single resources. Bummed code possesses a certain aesthetic quality, not unlike the compressed quality without a name. The indirectness of such code is pleasing to admire, though not, perhaps, to modify.

I still can't tell you what the quality is, but I can tell you some things about software that possesses it:

- It was not written to an unrealistic deadline.
- Its modules and abstractions are not too big—if they were too big, their size and inflexibility would have created forces that would overgovern the overall structure of the software; every module, function, class, and abstraction is small and named so that I know what it is without looking at its implementation.
- Any bad parts were repaired during maintenance or are being repaired now.
- If it is small, it was written by an extraordinary person, someone I would like as a friend; if it is large, it was not designed by one person, but over time in a slow, careful, incremental way.
- If I look at any small part of it, I can see what is going on—I don't need to refer to other parts to understand what something is doing. This tells me that the abstractions make sense for themselves—they are whole.
- If I look at any large part in overview, I can see what is going on—I don't need to know all the details to get it.
- It is like a fractal, in which every level of detail is as locally coherent and as well thought out as any other level.
- Every part of the code is transparently clear—there are no sections that are obscure in order to gain efficiency.
- Everything about it seems familiar.
- I can imagine changing it, adding some functionality.
- I am not afraid of it, I will remember it.

I wish we had a common body of programs with the quality, because then we could talk about them and understand. As it is, programs are secret and protected, so we rarely see any but those we write ourselves. Imagine a world in which houses were hidden from view. How would Alexander have found the quality with no name?

Think about the quality without a name when you look at your software. Do you think your software possesses it? What would you do to make your software have it?

Pattern Languages

Christopher Alexander's work is based on the premise that the quality without a name is an objective characteristic of things and places. As an architect, Alexander wants to know where this quality comes from and, more important, how to create it, how to generate it. In the previous essay, "The Quality Without a Name," we learned of the divorce centuries ago of beauty from reality. That science could survive the divorce is understandable because science seeks to *describe* reality. Science can live and succeed a long time before it needs to concern itself with describing what makes something beautiful— when something is contingent, as beauty seems to be in modern science, there is little need to describe it. Art, on the other hand, cannot ignore beauty or the quality without a name because artists *create* things—paintings, sculptures, buildings—that are beautiful, that have the quality without a name. There are few fields that blend art and science: Architecture is one, and computer science is another. Architects must design buildings that can be built and architects have a "theory" about what they do—at least architects like Alexander do. In computer science we can describe theories of software, and we create software.

In a field that combines art and science, its practitioners are often divided into camps—one that cares about the raw science and traditionally objective characteristics of the field and another that cares about the beauty and elegance of its creations. In software we care a lot about good code, and we rave about the good coders that we know. In doing this we fall into a trap that Alexander wants us to avoid: separating design from construction.

Before we plunge into patterns, I want to set the stage for the accepted view of how architecture is done, at least in the pre-Alexander world. Architects are hired to solve the problem of how to construct a building or buildings that meet certain constraints as specified by the future inhabitants or people in a position to specify what those inhabitants will need. The architect generally interviews those future inhabitants, studies the site, refers to the local building code, observes neighboring buildings, considers the building materials and construction protocols in the area, and then is inspired, in the manner of all artists, to create a set of drawings which a general contractor, in occasional conference with the architect, reduces to a physical object.

This view of architecture should be familiar to software theorists: It corresponds to the Waterfall Model with a Niagara-sized waterfall. The analyst or designer studies requirements, construction techniques (choice of language), and the context (related software) and, in the manner of all artists, creates the design which he or she happily tosses over a large wall for someone else to implement.

Enlightened folks nowadays tend to view askance this methodology, preferring one or another variant of the Spiral Model. (To be fair, the original Waterfall Model contains backward arrows linking later stages to earlier ones in a feedback type of arrangement, so there actually is not as sharp a contrast between the Waterfall and other models as some would have us believe.)

In fields separated into theory and practice we frequently find that aesthetics are fragmented. The theorist is often interested in beauty of a nature akin to mathematical beauty. In computer science such beauty is exactly mathematical beauty—Occam's razor slicing away fat, eliminating ornamentation, the hard chill beauty of a compact, precisely clever theorem. In architecture the architect works with drawings and models. The beauty of a drawing is not the beauty of a building. A beautiful drawing has spare lines, simple geometric shapes. A home or town is involved, ornamented, shows a care for human interests. Only an inhabitant could know what really counts.

The first of Alexander's contributions to architecture was to reject the separate architect and builder model and to posit user-centered design—in which users (inhabitants) design their own buildings—and the architect-builder who would blend the activities of design and construction. This was viewed by architects as high treason enough, let alone the curious quest for an elusive quality that Alexander cannot name.

The mechanism he proposed to accomplish his new model was the *pattern language*. A pattern language is a set of patterns used by a process to generate artifacts. These artifacts can be considered complexes of patterns. Each pattern is a kind of rule that states a *problem* to be solved and a *solution* to that problem. The means of designing a building, let's say, using a pattern language is to determine the most general problem to be solved and to select patterns that solve that problem. Each pattern defines subproblems that are similarly solved by other, smaller patterns. Thus we see that the solution to a large problem is a nested set of patterns.

Of course, because usually several patterns can solve a given problem, and any pattern requires a particular context to be effective, the process is generally not linear but is a sort of constraint-relaxation process. These days we know all about this sort of process, but back when Alexander came up with pattern languages, it was relatively new. His initial work was done when Noam Chomsky's transformational grammar was first in vogue, and the idea of a human activity—especially a creative one—being subject to rules and generated by a language was hard to swallow.

One interesting aspect of this approach is that it isn't grounded ultimately in *things* but stays at the level of patterns throughout. Some patterns talk about particular construction materials, but rarely is anything like a kitchen or a toilet mentioned. This is partly to maintain a level of elegance—there are only patterns and patterns of patterns, not things and patterns mixed—but also because the patterns talk, in addition, about social and human activities. For example, a stove is really a relationship between a surface, heating elements, thermostats, and switches, and a stove is part of a pattern of activity—cooking a meal, perhaps in a social context. Viewed this way, a stove is also a pattern within a larger pattern which is the whole kitchen whose other patterns involve refrigerators, preparation surfaces, cupboards, utensils, dining areas, walls, natural light sources, gardens, doors, and possibly much, much more. Some of the patterns must be implemented physically near one other, adding a layer of constraints on the solution to the kitchen design/construction process.

The way Alexander motivates pattern languages is with the example of barns. A farmer in a particular Swiss valley wishes to build a barn. Each barn has a double door to accommodate the haywagon, a place to store hay, a place to house the cows, and a place to put the cows so they can eat the hay; this last place must be convenient to the hay storage location so it is easy to feed the cows. There must be a good way to remove the cow excrement, and the whole building has to be structurally sound enough to withstand harsh winter snow and wind.

If each farmer were to design and build a barn based on these functional requirements, each barn would be different, probably radically different. Some would be round, the sizes would vary wildly, some would have double naves, doubly pitched roofs.

But barns in Swiss valleys do not vary wildly, so each farmer must be copying something. The farmer is not copying a specific other barn, because the barns do vary somewhat. Each is a little different because of where it is located and each farmer's particular needs. Therefore, farmers do not copy particular barns. Alexander says that each farmer is copying a set of patterns which have evolved to solve the Swiss-valley-barn problem. Barns in any given Swiss valley are similar, as are all alpine barns. They also are similar to barns in other areas of the world—a family resemblance among buildings—but they do have differences. For example, California barns are generally larger although they share the same general shape.

Here, again, is my working definition of abstraction:

> Abstraction *in programming is the process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use.* (Balzer et al. 1989)

Patterns in Alexander's sense are a lot like the common patterns but without the systematic variations. For Alexander, the variations of a pattern depend on the other patterns it contains and those containing it, thus eliminating the possibility of systematic variations. Patterns are not well structured enough to have systematic variations—their variations are too context dependent.

This is the format of patterns as presented in *A Pattern Language* (Alexander 1977b):

> *First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern by explaining how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. It describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution—the heart of the pattern—which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction—so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram, which shows the solution in the form of a diagram, with labels to indicate its main components.*
>
> *After the diagram, another three diamonds, to show that the main body of the pattern is finished. And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete the pattern, to embellish it, to fill it out.*
>
> *There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of . . . patterns as a whole, as a language within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it.* (Alexander 1977b)

Alexander's book contains 253 patterns covering regions, cities, towns, neighborhoods, transportation, homes, offices, work communities, relaxation areas, rooms, lighting, windows, gardens, waiting rooms, terraces, walls, building materials, and construction. One thing that strikes many readers of the pattern language is the degree to which Alexander talks about people and their activities in the patterns—the patterns are a response to his arguments about how life is best lived. Because of this, many regard *A Pattern Language* as a manual of how to live and how physical surroundings can support and enhance living.

The book is full of evocative black-and-white photographs of towns, parts of homes, scenes of life in Europe, Greece, the Middle East, Asia—all over—each demonstrating a place with the quality without a name and which the patterns are intended to create.

Alexander says:

> *And yet, we do believe, of course, that this language which is printed here is something more than a manual, or a teacher, or a version of a possible pattern language. Many of the patterns here are archetypal—so deep, so deeply rooted in the nature of things, that it seems likely that they will be a part of human nature, and human action, as much in five hundred years, as they are to-day. . . .*
>
> *In this sense, we have also tried to penetrate, as deep as we are able, into the nature of things in the environment. . . .*
> (Alexander 1977b)

Let me present an example pattern—I'll condense it quite a bit to save space.

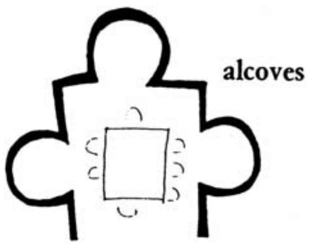
> *179. Alcoves***
>
> *. . . many large rooms are not complete unless they have smaller rooms and alcoves opening off them. . . .*
>
> ✣ ✣ ✣
>
> *No homogeneous room, of homogeneous height, can serve a group of people well. To give a group a chance to be together, as a group, a room must also give them the chance to be alone, in one's and two's in the same space.*
>
> *This problem is felt most acutely in the common rooms of a house—the kitchen, the family room, the living room. In fact, it is so critical there, that the house can drive the family apart when it remains unsolved. . . .*
>
> *In modern life, the main function of a family is emotional; it is a source of security and love. But these qualities will only come into existence if the members of the house are* physically able to be together as a family.

This is often difficult. The various members of the family come and go at different times of day; even when they are in the house, each has his own private interests. . . . In many houses, these interests force people to go off to their own rooms, away from the family. This happens for two reasons. First, in a normal family room, one person can easily be disturbed by what the others are doing. . . .Second, the family room does not usually have any space where people can leave things and not have them disturbed. . . .

To solve the problem, there must be some way in which the members of the family can be together, even when they are doing different things.

Therefore:

Make small places at the edge of any common room, usually no more than 6 feet wide and 3 to 6 feet deep and possibly much smaller. These alcoves should be large enough for two people to sit, chat, or play and sometimes large enough to contain a desk or table.



✣ ✣ ✣

Give the alcove a ceiling which is markedly lower than the ceiling height in the main room. . . . (Alexander 1977b)

What's interesting to me is the argument about how families are served by alcoves. This seems to go well with Alexander's desire to emphasize that the quality without a name is concerned with life and living. In fact, he says:

You see that the patterns are very much alive and evolving. (Alexander 1977b)

Part of Alexander's research program is to creative *generative* patterns—patterns that generate the quality without a name. Generativeness is an interesting trait. Typically something is said to be generative when it produces the generated quality indirectly.

A good example of a generative process is a random-number generator. Such programs perform simple arithmetic operations on simple starting quantities and produce a stream of numbers with have no simple relation to one other. If we look inside the generator, we don't see any structure or parts that are clearly related to the purpose of generating random numbers.

Another example from a different domain is advice on how to hit a tennis ball. The advice I'm thinking of is that you should not concentrate on hitting the ball at the point of impact but, instead, on hitting a point beyond the ball in the direction the racket is moving. The purpose of this advice is to avoid the effects of the muscles trying to slow down or stop at the point of impact. That is, if I ask you to hit a particular thing with a racket, your muscles will propel the racket toward the target, and just before hitting the target, the muscles controlling opposing movement will contract slightly in order to decelerate to the point of impact—though you will not, obviously, be decelerating to zero velocity, you will decelerate a little. The result of this small deceleration is to cause the racket to jiggle or jitter and for the impact to be less precise. If, on the other hand, you are told to hit something just beyond the target, your muscles will not involuntarily contract until you have hit the target, and as a result, the hit and trajectory will be smoother.

This is the same advice given to martial arts students who are attempting to break boards and bricks.

Such advice is generative: The goal is to hit smoothly and with full power, but this goal is not part of the advice. Rather, the advice is to do something else which has the side effect of achieving the goal.

Patterns in a pattern language are intended to be generative—they are supposed to generate the quality without a name. Just as the advice to write clearly and simply is not possible to follow—because writing clearly and simply is achieved by choosing words and syntax carefully, by choosing presentation order, and by deciding how to emphasize topics in sentences and paragraphs—neither is the advice to produce the quality without a name.

❧ ❧ ❧

The question we face is how pattern languages apply to software development. People who have tried to apply pattern languages to software have done what you might describe as the obvious thing: They started to develop patterns that are a prescription of how to solve particular problems that come up in development. This isn't new. For example, Knuth's *The Art of Computer Programming* (1969) is a multivolume book that contains exactly such information. This book presents programming situations and problems and describes, in the manner of a textbook, various solutions. A large consideration for Knuth is the performance—time and space complexity—of solutions, and he treats performance very mathematically.

Patterns provide several benefits to programmers and system developers. One is a common language. I've heard many discussions about programs in which the common reference points are algorithms and data structures Knuth describes—and the discussions frequently refer to such algorithms or data structures by citing Knuth's name for them and the volumes in which the descriptions can be found.

Another benefit is a common base for understanding what is important in programming. That is, every pattern in a program is a signpost that helps developers new to a program rapidly understand it. Each common pattern is both important and points out its important subpatterns. For example, in low-level programming languages, people often write `while` loops with the test at the end. When you first come across this style of loop, you wonder what it's all about—your ability to understand a program is challenged by the puzzle of why this loop is coded this peculiar way. Well, most people figure it out sooner or later or at least come to understand the purpose of the code. However, they can understand it more rapidly if they have read Knuth's `while`-loop optimization discussion—with such knowledge they can instantly recognize the Knuth `while` loop, and it is a matter of recognizing a pattern. The loop pattern is important, and the pattern highlights the meat of the loop—the part that is repeated—along with the loop test.

A third benefit is that with a corpus of patterns a programmer is able to solve problems more rapidly by having available a storehouse of solutions—a cookbook, if you will.

Among the folks who write software patterns is the Hillside Group. This group is at least spiritually led by Kent Beck and has met once or twice to talk about patterns and to gather them up—it's a sort of a writers workshop group for writing patterns.

This group, like many others concerned with patterns, focuses on objects and object-oriented programming. The idea is that behavior in object-oriented programs comes largely from configurations of objects and classes sending messages to one another according to protocols. The analogy to architecture should be plain.

Let's look at a pattern discussed by this group. I will, again, abbreviate it.

> Pattern: *Concrete Behavior in a Stateless Object*
>
> Context: *You have developed an object. You discover that its behavior is just one example of a family of behaviors you need to implement.*
>
> Problem: *How can you cleanly make the concrete behavior of an object flexible without imposing an unreasonable space or time cost, and with minimal effect on the other objects in the system?*
>
> Constraints: *No more complexity in the object. . . . Flexibility—the solution should be able to deal with system-wide, class-wide, and instance-level behavior changes. The changes should be able to take place at any time. . . . Minimal time and space impact. . . .*

Solution: *Move the behavior to be specialized into a stateless object which is invoked when the behavior is invoked.*

Example: *The example is debug printing. . . .* (Kent Beck, personal communication 1993)

The example actually explains the solution a lot better than does the solution description—the example is given in Smalltalk. The idea is that you define a side object (and a class) that has the behavior you want by defining methods on it. All the methods take an extra argument which is the real object on which to operate. Then you implement the desired behavior on the original object by first sending a message to `self` to determine the appropriate side object and then sending the side object a message with the real object as an extra argument. By defining the method that returns the side object you can get either instance-level, class-level, or global changes in behavior.

This is a useful pattern, and you can imagine how a book full of stuff like this would turn a smart but inexperienced object programmer into something a little closer to an expert.

Patterns seem to be a partial solution to the overabstraction problem I talked about in "Abstraction Descant." They are a way to take advantage of common patterns without building costly, confusing, and unnecessary abstractions when the goal is merely to write something understandable. That is, when there are more idioms to use, using them is far better than inventing a new vocabulary. There are lots of reasons that abstractions are used, and I'm not saying we shouldn't use them, but let's not confuse their convenience in some situations as proof of their unqualified usefulness. For example, abstractions allow us to avoid typing (fingers pounding the keyboard). However, avoiding typing is something a programming environment is supposed to help accomplish, and one can imagine a programming environment that would let people use a pattern about as easily as an ordinary abstraction. Keep in mind that the most useful patterns are quite large, like the one regarding concrete behavior in stateless objects. Furthermore, as Alexander points out, patterns interact with larger and smaller patterns in such a way that the actual manifestation of any given pattern is influenced by and influences several or many other patterns.

Alexander talks about the poetry of patterns. Poetry is at least partly distinguished from prose by its stronger use of *compression*. Compression is that characteristic of a piece in some language in which each word assumes many meanings and derives its meaning from the context. In this way, a small expression can perform a lot of tasks because the context is rich. This is one reason that inheritance works and is also a way that inheritance can go wrong. When you write a subclass, you can reuse the methods above it—and the code you write is compressed because it takes its meaning from the context of its superclasses. This is good, but if you have a deep or dense hierarchy, you could easily spend a lot of time understanding it in order to write correct, highly compressed code.

For patterns, you can have compression if several patterns are present in one space. Alexander says:

> *It is quite possible that all the patterns for a house might, in some form, be present, and overlapping, in a simple one-room cabin. The patterns do not need to be strung out, and kept separate. Every building, every room, every garden is better, when all the patterns which it needs are compressed as far as it is possible for them to be. The building will be cheaper; and the meanings in it will be denser.*
> (Alexander 1977a)

Compression is what we saw when we looked at the constraints of European and some Third-World architecture: small scale. With limited space, all the features that people require to live must be present in the space available. This forces people to be clever to design in all the parts in minimal space, most likely by overlapping and causing this poetry—the poetry of compression. It also forces things into small, irregular pieces.

The question is: Is software that is compressed better than software that is not? I think there is a great deal of admiration for compressed software, but is it more maintainable?

❧ ❧ ❧

Patterns in software are a good idea, but I have rarely seen the attention to the human side—to living within software—that I see in Alexander's patterns. One architect commented on Alexander that he regarded the patterns as providing information, sort of heuristics, about architecture, and I have to admit the patterns I've seen about software do that—they provide information that the software practitioner just plain ought to know.

But the picture is not quite complete, and I fear it may never be. The use of patterns is clearly helpful, if for no other reason to capture in a common format programming lore and tips—a sort of better-organized *Art of Computer Programming*. The "but" concerns the quality without a name. Alexander hoped that his pattern language would generate towns, buildings, communities, homes, offices, gardens, and places with that quality. And he had the benefit of knowing what he was trying to generate. He could go to Florence and walk the piazzas, the colonnades, the small hidden gardens, he could visit the temples in Kyoto, the perfect alcoves in Norway, white hilltowns of Greece. We as computer scientists do not have examples of programs exhibiting the quality without a name that we all can agree on or even just name. That's step 1. Step 2 is to understand the quality. I took a stab at it in “The Quality Without a Name,” but I admit I was simply guessing.

The worst thing is this: Alexander had the chance in the 1970s to try out his pattern language. He observed the results of others trying it out and he even tried it out himself several times. And guess what: It didn’t work. In one project in Mexicali, Alexander, his colleagues, students, and a community of people constructed a community. Alexander says he found hints of the quality in only a few places. Otherwise, the houses were “funky”. That is, when Alexander tried out his theory, it failed.

The story is not over.

The Failure of Pattern Languages

We have been exploring the Christopher Alexander saga and it might surprise some to learn that Alexander completed by the early 1970s all the work I've reported so far—in what some would call the infancy of computer science. People have read his work and taken off from it—The Hillside Group and others are writing software patterns, believing they are doing something worthwhile—and they are. But are they doing for software what Alexander set out to do for architecture—find a process to build artifacts possessing the quality without a name? Or is the quality unimportant to what the software pattern writers are doing? Is it important only that they are accomplishing something good for software, and Alexander's original goal is unimportant, merely a catalyst or inspiration—the way fine drizzle drawing his eyes low, narrowing streetlights to a sheltering fog, can inspire a poet to write a poem of intimacy and its loss? Or the way an inept carpenter building an overly sturdy birdcage can inspire another person to construct a tiger's cage?

I think the quality without a name is vital to software development, but I'm not yet sure how, because I am not clear on what the quality without a name is in the realm of software. It sits zen in the midst of a typhoon frenzy of activity in both architecture and software. Alexander's story does not end with the publication of *A Pattern Language* in 1977. It went on and still goes on. And Alexander did not sit still after he wrote the patterns. Like any scientist he tried them out.

And they did not work.

Read Alexander's own words:

> *All the architects and planners in christendom, together with* The Timeless Way of Building (Alexander 1979) *and the* Pattern Language (Alexander 1977a), *could still not make buildings that are alive because it is other processes that play a more fundamental role, other changes that are more fundamental.* (Grabow 1983)

Alexander reached this conclusion after completing some specific projects. One was the Modesto Clinic. In this project an architect from Sacramento used Alexander's pattern language to design and build a medical clinic in Modesto (in the Central Valley of California). The building was a success in the sense that a building was actually constructed and its plan looked good on paper. Alexander noted:

> *Up until that time I assumed that if you did the patterns correctly, from a social point of view, and you put together the overall layout of the building in terms of those patterns, it would be quite alright to build it in whatever contemporary way that was considered normal. But then I began to realize that it was not going to work that way.* (Grabow 1983)

Even though the Sacramento architect tried hard to follow the patterns, the result was dismal. Alexander says about the clinic:

> *It's somewhat nice in plan, but it basically looks like any other building of this era. One might wonder why its plan is so nice, but in any really fundamental terms there is nothing to see there. There was hardly a trace of what I was looking for.* (Grabow 1983)

This wasn't an isolated failure, but one repeated frequently by other people trying to use the pattern language from bootlegged copies of *A Pattern Language*:

> *Bootleg copies of the pattern language were floating up and down the West Coast and people would show me projects they had done and I began to be more and more amazed to realize that, although it worked, all of these projects basically looked like any other buildings of our time. They had a few differences. They were more like the buildings of Charles Moore or Joseph Esherick, for example, than the buildings of S.O.M. or I. M. Pei; but basically, they still belonged perfectly within the canons of mid-twentieth century architecture. None of them whatsoever crossed the line.* (Grabow 1983)

Alexander noticed a more bizarre phenomenon than the fact that the buildings were no different from their contemporaries—the architects believed they were different, vastly and remarkably different. Alexander said:

> *They thought the buildings were physically different. In fact, the people who did these projects thought that the buildings were quite different from any they had designed before, perhaps even outrageously so. But their perception was incredibly wrong; and I began to see this happening over and over again—that even a person who is very enthusiastic about all of this work will still be perfectly capable of*

> *making buildings that have this mechanical death-like morphology, even with the intention of producing buildings that are alive.*
>
> *So there is the slightly strange paradox that, after all those years of work, the first three books are essentially complete and, from a theoretical point of view, do quite a good job of identifying the difference but actually do not accomplish anything. The conceptual structures that are presented are just not deep enough to actually break down the barrier. They actually do not do anything.* (Grabow 1983)

Alexander determined that they failed because the geometry of the buildings was not as different from the standard modern geometry as it needed to be to generate the quality. One of his reactions was to consider the *process* of building: the mortgage process, the zoning process, the construction process, the process of money flowing through the system, the role of the architect, and the role of the builder. By controlling the process, you control the result, and if the control retains the old, broken process, the result will be the old, broken architecture.

This resonates with what we see in software development: The structure of the system follows the structure of the organization that put it together, and to some extent, its quality follows the nature of the process used to produce it. The true problems of software development derive from the way the organization can discover and come to grips with the complexity of the system being built while maintaining budget and schedule constraints. It is not common for organizations to try to put together a novel large artifact, let alone doing it on schedule. When an engineering team designs and builds a bridge, for example, it is creating a variant of a well-known design, and so many things about that design are already known that the accuracy of planning and scheduling depends on how hard the people want to work, not on whether they can figure out how to do it.

Even while considering process, Alexander never lost sight of geometry.

> *[T]he majority of people who read the work, or tried to use it, did not realize that the conception of geometry had to undergo a fundamental change in order to come to terms with all of this. They thought they could essentially graft all the ideas about life, and patterns, and functions on to their present conception of geometry. In fact, some people who have read my work actually believe it to be somewhat independent of geometry, independent of style—even of architecture.* (Grabow 1983)

From the time when buildings routinely possessed the quality without a name to the present when almost no buildings possess that quality, the process of building has changed from the so-called renaissance building paradigm to the current industrial building paradigm, from one in which building was a decentralized activity in which the architect, if one was used, was closely associated with the building or even participated in the building, and material was handcrafted on site, to one in which an architect might design hundreds of homes which were then built by a contractor buying and using modular parts, perhaps minimally customizing them—for example, cutting lumber to the proper length. At the same time, because building became a high-volume activity, the process of funding building changed so that, now, a large sum of money needs to be allocated to produce any homes at all.

As I mentioned in "Habitability and Piecemeal Growth," one problem with the building process is *lump-sum development*. In such development few resources are brought to bear on the problems of repair and piecemeal growth. Instead, a large sum of money is dedicated to building a large artifact, and that artifact is allowed to deteriorate somewhat, and anything that is found lacking in the design or construction is ignored or minimally addressed until it is feasible to abandon the building and construct a replacement. This phenomenon also occurs in the mortgage process. The bank lends someone, say a developer, a large sum of money to construct a home. A homebuyer purchases this home by assuming a large debt. The debt is paid off over time, with the early payments dedicated mostly to paying the interest, which accumulates, and only at the end of the mortgage period is the principal taken down. The result is that a homeowner might pay $1 million for a house that cost $400,000 to build. The problem with this—aside from all the problems you can easily identify yourself—is that the additional $600,000 paid for the house is not available for repair and piecemeal growth. It is a fee to the bank. And the house is not improved in any way or only minimally during the payment period—10 to 30 years—at which point the house is sold to someone else who pays another (or the same!) bank another enormous fee. The key ingredient to long-term development—piecemeal growth—is thwarted.

Alexander started a lengthy process himself of constructing arguments to show that process itself was the root cause for the practical failure of his theory of the quality without a name and of his particular pattern language—that the process of construction encompassed so many things controlling geometry that the outcome had to be as flawed and disappointing as what he saw in the early, uncontrolled experiments.

First he needed to convince himself that process could be the primary determiner of the outcome of a generative process. He got this from D'Arcy Thompson; Alexander said:

> *What Thompson insisted on was that every form is basically the end result of a certain growth process. When I first read this I felt that of course the form in a purely static sense is equilibrating certain forces and that you could say that it was even the product of those forces—in a non-temporal, non-dynamic sense, as in the case of a raindrop, for example, which in the right here and now is in equilibrium with*

> *the air flow around it, the force of gravity, its velocity, and so forth—but that you did not really have to be interested in how it actually got made. Thompson however was saying that everything is the way it is today because it is the result of a certain history—which of course includes how it got made. But at the time I read this I did not really understand it very well; whereas I now realize that he is completely right.* (Grabow 1983)

It's somewhat amazing that Alexander would fail to understand this right off, because his theory—patterns generating the quality without a name—is an example of it. His error, if there was one, was not to go far enough with his theory. Once he latched onto this insight he went hog wild. First he examined the process closely, identifying places where its details did not serve the quality without a name and geometry. Second, he performed several important experiments in which he controlled or nearly controlled the entire process.

These experiments were based on an alternative process developed by Alexander, which he called the "grassroots housing process." The basic idea is that a sponsor—a group of people, a corporation—would provide land at a reasonable price. There would be a builder who was actually an architect, a builder, and a manager rolled into one. Families would get an allotment of money to begin construction. The builder would help, and with the pattern language each family would build its own home. Each family would pay a fee per year with the following characteristics. The fee would be based on square footage and would decline from a very high rate in the early years to a very low one in later years. It was assumed to take around 13 years to pay off the fee. Materials for building would be free to families (of course, paid for by the fees). This means that families would be encouraged initially to build small homes. Because materials would be free and the only fees would be for square footage, each family would be encouraged to improve or embellish its existing space and the cluster's common space. As time passed and the fees dropped in later years, homes could be enlarged. These clusters would nest in the sense that there would be a larger "political" unit responsible for enhancing structures larger than any particular cluster. For example, roads would be handled in this way and the political unit would be a sort of representative government.

The existence of free materials and nested clusters would, Alexander hoped, create a mechanism for planning within a community and with a nearby or enclosing community.

The builder would help the families do their own building by instruction or by doing those jobs requiring the most skill. Each builder would have several apprentices who would be trained on the job. A builder would work with a cluster of families, and over time the builder would gradually move on to service another cluster, at which point the first cluster would become self-sufficient.

The way this scheme would work, of course, is the same way the banks work, but with a lower fee and with the community—the cluster—acting as the bank. Profits from the process would be used to sponsor other clusters.

After presenting this proposed process, Christopher Alexander was shocked to be quizzed about his views on Marxism. Nevertheless, two important concepts came out of it: the nested cluster with shared common areas and the architect-builder.

This led to several projects. One was to see whether local politics could be changed to support this new building process. To this end Alexander managed to get passed a Berkeley referendum in the early 1970s that put a moratorium on new construction and established a commission to look into a new master plan based on participatory planning of the sort talked about in *The Oregon Experiment* (Alexander 1975). The result was not quite what he had in mind: There was a new master plan, but one that asked local neighborhoods which streets could be closed. As a result, some of them were—if you drive through Berkeley today, you can still experience the results.

The most ambitious experiment was to build a community in Mexicali. The Mexican government became convinced that Alexander would be able to build a community housing project for far less than the usual cost, so they gave him the power he felt he needed to organize the project. The land was provided in such a way that the families together owned the encompassed public land and each family owned the land on which their home was built. The point of the experiment was to see whether with a proper process and a pattern language, a community could be built that demonstrated the quality without a name. Because of the expected low cost of the project and the strong recommendation of the University of Mexico regarding Alexander's work, the Mexican government was willing to allow Alexander to put essentially into practice his grassroots system of production. The details of this system hinged on the answers to these questions (all seven questions are from Alexander 1985):

1. *What kind of person is in charge of the building operation itself?*

   An architect-builder is in charge. This corresponds to the master architect of a software system who also participates in coding and helps his codevelopers with their work.

2. *How local to the community is the construction firm responsible for building?*

   Each site has its own builder's yard, each responsible for local development. This corresponds to putting control of the computer and software resources for each small project within that project. Local control and physically local resources are important.

3. *Who lays out and controls the common land between the houses, and the array of lots and houses?*

   This is handled by the community itself, in groups small enough to come to agreement in face-to-face meetings. This corresponds to small group meetings to discuss and negotiate interfaces in a project. There is no centralized decision maker, but a community of developers sits down and discusses in the group the best interfaces.

4. *Who lays out the plans of individual houses?*

   Families design their own homes. This corresponds to each developer designing his or her own implementations for a component.

5. *Is the construction system based on the assembly of standard* components, *or is it based on acts of creation which use standard* processes*?*

   Construction is based on a standard process rather than by standard components. This goes against one of the supposed tenets of the object philosophy in which standardized class libraries are *de rigueur*. Nevertheless, many experiences show that the true benefits of reuse come from reuse within a project and not as much from among projects. Such successful reuse is based on being able to model the domain of the project so that classes defined to represent aspects of the domain can be used for several purposes within that same model. Typically such a model is called a *framework*.

6. *How is cost controlled?*

   Cost is controlled flexibly so that local decisions and trade-offs can be made. This corresponds to giving a project a total budget number rather than breaking it down too far, such as one budget for hardware and another for developers.

7. *What is the day-to-day life like, on-site, during the construction operation?*

   It is not just a place where the job is done but a place where the importance of the houses themselves as homes infuses the everyday work. Developers need to have their own community in which both the work and the lives of the developers are shared: meals, rest time together in play. It's called team building in the

> management literature, but it's more than that, and every development manager worth paying knows that this is one of the most important parts of a project.

You might wonder why Alexander (or I, for that matter) considers a project's day-to-day life to be important. It's because the quality of the result depends on the quality of communication between the builders or developers, and this depends on whether it is fun to work on the project and whether the project is important to the builder or developer in a personal sense and not just in a monetary or job sense.

Alexander tells the story of this project—including the sorts of meals the families had and their celebrations after completing major projects—in *The Production of Houses* (1985). At the end of that book Alexander tells about the failures of the project as seen close up—that is, before he was able to sit back and look at the results objectively. He says there were a number of failures. First, the Mexican government lost faith in the project and pulled the plug after five houses were built, leaving 25 unbuilt. Partly they lost faith because the buildings looked "traditional" rather than modern. Alexander said:

> *The almost naïve, childish, rudimentary outward character of the houses disturbed them extremely. (Remember that the families, by their own frequent testimony, love their houses.)* (Alexander 1985)

Another reason was that the government was dismayed by Alexander's experimentation with construction systems. The government felt that because Alexander was a world authority on architecture and building, he would simply apply what he knew to produce beautiful homes cheaply and rapidly.

One failure that seemed to disturb Alexander was that the builder's yard was abandoned within three years of the end of the project. He felt that the process would have continued without his involvement once the families saw that they could control their own living spaces.

But did the buildings and the community have the quality without a name? Alexander said at the time:

> *The buildings, for example, are very nice, and we are very happy that they so beautifully reflect the needs of different families. But they are still far from the limpid simplicity of traditional houses, which was our aim. The roofs are still a little awkward, for example. And the plans, too, have limits. The houses are very nice internally, but they do not form outdoor space which is as pleasant, or as simple, or as profound as we can imagine it. For instance, the common land has a rather complex shape, and several of the gardens are not quite in the right place. The freedom of the pattern language, especially in the*

> *hands of our apprentices, who did not fully understand the deepest ways of making buildings simple, occasionally caused a kind of confusion compared with what we now understand, and what we now will do next time.* (Alexander 1985)

This chilling reference to the deep understanding required to build buildings with the quality without a name is echoed in the discussion of the builder's yard:

> *When their [the government's] support faded, the physical buildings of the builder's yard had no clear function, and, because of peculiarities in the way the land was held, legally, were not transferred to any other use, either; so now, the most beautiful part of the buildings which we built stand idle. And yet these buildings, which we built first, with our own deeper understanding of the pattern language, were the most beautiful buildings in the project. That is very distressing, perhaps the most distressing of all.* (Alexander 1985)

Later, after some reflection Alexander became more harsh:

> *There was one fact above everything else I was aware of, and that was that the buildings were still a bit more funky than I would have liked. That is, there are just a few little things that we built down there that truly have that sort of limpid beauty that have been around for ages and that, actually, are just dead right. That's rare; and it occurred in only a few places. Generally speaking, the project is very delightful—different of course from what is generally being built, not just in the way of low-cost housing—but it doesn't quite come to the place where I believe it must.*
>
> *. . . But what I am saying now is that, given all that work (or at least insofar as it came together in the Mexican situation) and even with us doing it (so there is no excuse that someone who doesn't understand it is doing it), it only works partially. Although the pattern language worked beautifully—in the sense that the families designed very nice houses with lovely spaces and which are completely out of the rubric of modern architecture—this very magical quality is only faintly showing through here and there.* (Grabow 1983)

Alexander noticed a problem with using the word *simplicity* to refer to the fundamental goal of the patterns:

> *We were running several little experiments in the builder's yard. There is an arcade around the courtyard with each room off of the arcade designed by a different person. Some of the rooms were designed by my colleagues at the Center and they also had this unusual funkiness—still very charming, very delightful, but not calm*

> *at all. In that sense, vastly different from what is going on in the four-hundred year old Norwegian farm where there is an incredible clarity and simplicity that has nothing to do with its age. But this was typical of things that were happening. Here is this very sort of limpid simplicity and yet the pattern language was actually encouraging people to be a little bit crazy and to conceive of much more intricate relationships than were necessary. They were actually disturbing. Yet in all of the most wonderful buildings, at the same time that they have all of these patterns in them, they are incredibly simple. They are not simple like an S.O.M. building;—sometimes they are incredibly ornate—so I'm not talking about that kind of simplicity. There is however a kind of limpidity which is very crucial; and I felt that we just cannot keep going through this problem. We must somehow identify what it is and how to do it—because I knew it was not just my perception of it.*
>
> *The problem is complicated because the word simplicity completely fails to cover it; at another moment it might be exactly the opposite. Take the example of the columns. If you have the opportunity to put a capital or a foot on it, it is certainly better to do those two things than not—which is different from what the modern architectural tradition tells you to do. Now, in a peculiar sense, the reasons for it being better that way are the same as the reasons for being very simple and direct in the spacing of those same columns around the courtyard. I'm saying that, wherever the source of that judgment is coming from, it is the same in both cases. . . . The word simplicity is obviously not the relevant word. There is something which in one instance tells you to be simple and which in another tells you to be more complicated. It's the same thing which is telling you those two things.* (Grabow 1983)

Another part of the problem Alexander saw with the Mexicali project had to do with the level of mastery needed in the construction process. At the start of the project he felt that merely having the same person do both the design and the construction was enough, so that the important small decisions dictated by the design would be made correctly. But during the project and later on Alexander learned that the builder needed more.

> *Only recently have I begun to realize that the problem is not merely one of technical mastery or the competent application of the rules—like trowelling a piece of concrete so that it's really nice—but that there is actually something else which is guiding these rules. It actually involves a different level of mastery. It's quite a different process to do it right; and every single act that you do can be done in that sense well or badly. But even assuming that you have got the technical part clear, the creation of this quality is a much more complicated*

> *process of the most utterly absorbing and fascinating dimensions. It is in fact a major creative or artistic act—every single little thing you do—and it is only in the years since the Mexican project that I have begun to see the dimensions of that fact.* (Grabow 1983)

Not only must you decide to build the right thing, but you also must build it with skill and artistry. This leads to an obvious question: Are buildings with the quality without a name just too hard to put together deliberately, and were they created in older societies only by chance and survive only because of the quality? Or even more cynically, is there a form of nostalgia at work in which everything old is revered, because it comes from a more innocent and noble age?

There is another, chilling possibility: Perhaps it takes a real artist to create buildings and towns possessing the quality without a name. If you think about it, however, there is nothing really surprising or shocking about this.

Not everyone can write a great poem or paint a picture that will last through the ages. When we go to museums, we see art with the quality without a name and we are not surprised. And when we go to an adult-education art class we are not surprised to see good art, but art that is a little funky and perhaps it's different but it's not great.

The question is whether it is possible to write down rules or patterns for architecture—and software and art—so that ordinary people can follow the rules or patterns and, by the nature of the patterns and using only the abilities of ordinary people, beauty is generated. If this happens, then the rules or patterns are generative, which is a rare quality.

Art teachers can provide generative rules. For example, my father watched and "studied" the various art programs on public television for 15 years, and I'll be darned but he stopped painting like a fifth grader waiting for recess and started being a pretty decent still-life and landscape painter. You won't see his work in a museum in 20 years, but if you saw it on the wall in my house, you wouldn't bat an eye.

What he learned about painting surprised me. I expected that the lessons would talk about drawing skills and how to reproduce what you saw in reality by attending to each detail in order to create an accurate reproduction. What he learned instead was that by using the nature of the medium, the paint, the brush, and brush strokes, he could create the same effects as those of reality through reflected light and by the oil on canvas. So, instead—as I expected—of spending hours painting each branch on a conifer, my father would dab and push with a thick brush to create the effect of needled branches, and from a proper viewing distance his conifers looked as real as real.

Let's recall what Alexander said about the people who helped build the houses in Mexicali and who live there: "Remember that the families, by their own frequent testimony, love their houses." Perhaps these houses did not live up to Alexander's high artistic standards, but they were nice homes and served their users well. Like my father's paintings, they are better than the untutored could accomplish. And there's nothing wrong with that.

Perhaps Alexander's pattern language is working as well as it can—it's just that the artistry required to make buildings having the quality without a name wasn't present in any of the families who worked with Alexander or even in Alexander's team. The failure of the pattern language in the bootleg-designed houses and in Mexicali are maybe simply the failures of artistry.

Perhaps, and perhaps these are true failures, but this should not dissuade us from considering how to create things with the quality without a name.

It is easy to see that the process of constructing buildings has an obvious correspondent in software: The process of software construction is the single most important determining factor in software quality. Alexander's building process adapted to software could be called a form of *incremental development* because it advises the use of small groups with autonomy, architect-builders at the helm.

But what of geometry? Alexander always goes back to this. And one of his key questions is: What dictates geometry possessing the quality without a name—what is that thing that is falsely called *simplicity*?

What corresponds to geometry for us?

I think it is the code itself. Many talk about the need for excellent interfaces and the benefits of separating interface from implementation so that the implementation may vary. But few people talk seriously about the quality of the code itself. In fact, most theorists are eager to lump it into the category of things best not discussed, something to be hidden from view so that it can be changed in private. But think of Alexander's remarks: The quality comes in nearly equal part from the artistry and creativity of the builder who is the one whose hands most directly form the geometry that gives the building its quality and character. Isn't the builder the coder? And isn't the old-style software methodology to put design in the hands of analysts and designers and to put coding in the hands of lowly coders, sometimes offshore coders who can be paid the lowest wages to do the least important work?

Methodologists who insist on separating analysis and design from coding are missing the essential feature of design: The design is in the code, not in a document or in a diagram. Half a programmer's time is spent exploring the code, not in typing it in or changing it. When you look at the code you see its design, and that's most of what you're looking at, and it's mostly while coding that you're designing.

> *Poincaré once said: "Sociologists discuss sociological methods; physicists discuss physics." I love this statement. Study of method by itself is always barren. . . .* (Alexander 1964)

This could be a lesson for some of our software development methodologists: Study software, not software methods.

Patterns can help designers and designer-coders to make sure they put the right stuff in their software, but it takes a coder with extraordinary mastery to construct software having the quality without a name.

And what of simplicity? Can it be that blind subservience to simplicity in software development can lead to the same “death-like morphology” that it causes in architecture?

In 1995, software developers are writing and publishing patterns, and the “patterns movement” is gaining momentum. What are they actually doing? Most of the patterns I’ve seen present solutions to technical problems. For example, a pattern language I’m familiar with explains how to produce a design for a program that will validate a complex set of input values presented on forms that the end user fills in. It’s a very nice pattern language, but I’m not sure where the quality without a name is in this language. If I needed to write a system with a component that had to validate input fields, I would use it, but I doubt that the quality would emerge from what I learned from the pattern language.

From the pattern language I would learn the issues that go into an input validation component, and I would be able to get all the parts right, and perhaps folks later on maintaining my component would have an easier job of it for my having used the pattern language, but I doubt there would be any artistic merit in it or the quality.

When I look at software patterns and pattern languages, I don’t see the quality without a name in them, either. Recall that Alexander said that both the patterns themselves and the pattern language have the quality. In many cases today the pattern languages are written quickly, sort of like students doing homework problems. I heard one neophyte pattern writer say that when writing patterns he just writes what he knows and can write four or five patterns at a sitting.

That patterns that I wrote neither possess the quality nor generate it is not surprising. Patterns are an art form, and they are designed to generate art within the domain about which they speak. Art is not created at the clip of four or five pieces per sitting. It takes time and the right context and the right preparation to write them.

My patterns are doggerel, just as are poems written quickly this way—four or five at a sitting. Poetic doggerel serves a purpose: therapy. And maybe so does pattern writing of the sort we’re talking about.

Patterns written in this way will almost never generate the quality without a name, and it is likely that people who write patterns like this and see them fail will attribute the failure to the concept of patterns and pattern languages. And people who use the patterns and fail may do so as well.

But Alexander's pattern language is not doggerel: It was written by many people over many years, and each was subject to extensive discussion and criticism before being published. It is clear that the patterns themselves have the quality without a name, and I think this is because Alexander and his colleagues are pattern artists.

It just seems that the experiments that Alexander's team did in Mexicali were not done by artists or that the muse didn't visit them during the process.

Alexander's story is nearly up to date, but to many it disappears at the point we've left him here. Our modern-day pattern writers are content to stop with *A Pattern Language* and *The Timeless Way of Building* and to go off writing patterns. They ignore the failures that Alexander himself saw in his own processes at that point. The next chapter of his story takes him back to geometry and art. It's probably a story that has only the most metaphorical application to software because it has to do with beads, beauty, art, geometry, and Turkish carpets.

The Bead Game, Rugs, and Beauty

People in software research continue to find inspiration in the work of Christopher Alexander—they find his concept of pattern language a means to codify design skills. Some of them, though, don't see the relevance of the quality without a name, and anyhow some doubt there is anything about older building styles than they have survived through time by a process of natural selection—the old buildings we see today in Europe, let's say, are nicer because the cruddy ones were torn down and the nice ones imitated.

To many software patterns folk the quality without a name doesn't apply to things like software anyhow. I agree that most software—at least the software I see—doesn't have such a quality, but does that mean it couldn't? I find it odd, though, to take so much inspiration from the simple, mechanical parts of a person's work—the form of the pattern language and terms like *forces*—but to ignore the heart of it. I'm not so sure the quality without a name is irrelevant.

When we last left Christopher Alexander, he had apparently despaired in his quest for the quality without a name based on his experiences with the Modesto and Mexicali projects. In those projects his pattern language was used to construct a clinic (Modesto) and a small community (Mexicali). In Modesto the architect in charge was not able to control the process of spending and building as Alexander advocated in his "grassroots housing process." In the Mexicali project, where Alexander was in charge and had a special arrangement with the Mexican government, he had all the control he needed. But the results were disappointing. Alexander felt the buildings were "funky" and only in places demonstrated a hint of the quality.

It is at this point that Alexander went off in search of a universal formative principle, a generative principle governing form that would be shared by both the laws of nature and great art. If the principle could be written down and was truly formative, then aesthetic judgment and beauty would be objective and not subjective, and it would be possible to produce art and buildings with the quality without a name.

If there were such a universal principle, any form that stirred us would do so at a deep cognitive level rather than at a representational level, where its correspondence to reality is most important. That is, the feeling great form in art gives us would be a result of the form operating directly on us and in us rather than indirectly through nature; and nature would share the same forms because the principle is universal. Many philosophers from Plato onward believed in this deeper level of form, including Alexander. According to Herbert Read,

> *The increasing significance given to* form *or* pattern *in various branches of science has suggested the possibility of a certain parallelism, if not identity, in the structures of natural phenomena and of authentic works of art. That the work of art has a formal structure of a rhythmical, even of a precisely geometrical kind, has for centuries been recognised by all but a few nihilists. That some at any rate of these structures or proportions—notably the Golden Section—have correspondence in nature has also been recognised for many years. The assumption, except on the part of a few mystics, was that nature in these rare instances, was paying an unconscious tribute to art; or that the artist was unconsciously imitating nature. But now the revelation that perception is itself a pattern-selecting and pattern-making function (a Gestalt formation); that pattern is inherent in the physical structure and functioning of the nervous system; that matter itself analyses into coherent patterns or arrangements of molecules; and that the gradual realisation that all these patterns are effective and ontologically significant by virtue of an organisation of their parts which can only be characterised as* aesthetic*—all this development has brought works of art and natural phenomena on to an identical plane of inquiry.* (Grabow 1983; Read 1951)

Alexander took this point of view seriously and proposed the "bead game conjecture," a mechanism that unifies all forms—science, art, music, society.

> *That it is possible to invent a unifying concept of structure within which all the various concepts of structure now current in different fields of art and science, can be seen from a single point of view. This conjecture is not new. In one form or another people have been wondering about it, as long as they have been wondering about structure itself; but in our world, confused and fragmented by specialisation, the conjecture takes on special significance. If our grasp of the world is to remain coherent, we need a bead game; and it is therefore vital for us to ask ourselves whether or not a bead game can be invented.* (Alexander 1968)

"Bead game" refers to Hermann Hesse's imaginary game in which all forms—art, science, nature—can be represented in a single way.

As I noted in "The Failure of Pattern Languages," Alexander attributed this failure of his pattern language to two things: The first was that the level of mastery of the pattern language and of the building and design skills needed to produce the quality without a name were in fact limited in the folks doing the work; and the second was that the participants—the architects and builders—did not sufficiently appreciate the geometrical aspects of beauty and the quality. Alexander said:

> *I had been watching what happens when one uses pattern languages to design buildings and became uncomfortably aware of a number of shortcomings. The first is that the buildings are slightly funky—that is, although it is a great relief that they generate these spontaneous buildings that look like agglomerations of traditional architecture when compared with some of the concrete monoliths of modern architecture, I noticed an irritatingly disorderly funkiness. At the same time that it is lovely, and has many of these beautiful patterns in it, it's not calm and satisfying. In that sense it is quite different from traditional architecture which appears to have this looseness in the large but is usually calm and peaceful in the small.*
>
> *To caricature this I could say that one of the hallmarks of pattern language architecture, so far, is that there are alcoves all over the place or that the windows are all different. So I was disturbed by that—especially down in Mexico. I realized that there were some things about which the people putting up the buildings did not know—and that I knew, implicitly, as part of my understanding of pattern languages (including members of my own team). They were just a bit too casual about it and, as a result, the work was in danger of being too relaxed. As far as my own efforts were concerned, I realized that there was something I was tending to put in it in order to introduce a more formal order—to balance this otherwise labyrinthine looseness.*
>
> *The other point is that even although the theory of pattern languages in traditional society clearly applies equally to very great buildings—like cathedrals—as well as to cottages, there was the sense that, somehow, our own version of it was tending to apply more to cottages. In part, this was a matter of the scale of the projects we were working on; but it also had to do with something else. It was almost as if the grandeur of a very great church was inconceivable within the pattern language as it was being presented. It's not that the patterns don't apply; just that, somehow, there is a wellspring for that kind of activity which was not present in either* A Pattern Language (1977a) *or* The Timeless Way of Building (1979). (Grabow 1983)

More important, I think, is the fact that people did not quite understand the geometrical nature of what Alexander was talking about. The geometrical nature of the quality is brought out in Chapter 26 of *The Timeless Way of Building*, called "Its Ageless Character." Let me quote some passages from that chapter to give you a flavor:

> *And as the whole emerges, we shall see it takes that ageless character which gives the timeless way its name. This character is a specific, morphological character, sharp and precise, which must come into being any time a building or a town becomes alive: it is the physical embodiment, in buildings, of the quality without a name. . . .*
>
> *In short, the use of languages does not just help to root our buildings in reality; does not just guarantee that they meet human needs; that they are congruent with forces lying in them—it makes a concrete difference to the way they look. . . .*
>
> *This character is marked, to start with, by the patterns underlying it. . . .*
>
> *It is marked by greater differentiation.*
>
> *If we compare these buildings [the ones with the quality without a name] with the buildings of our present era, there is much more variety, and more detail: there are more internal differences among the parts.*
>
> *There are rooms of different sizes, doors of different widths, columns of different thickness according to their place in the building, ornaments of different kinds in different places, gradients of window size from floor to floor. . . .*
>
> *The character is marked, in short, by greater differences, and greater differentiation.*
>
> *But it is marked, above all, by a special balance between "order" and "disorder".*
>
> *There is a perfect balance between straight lines and crooked ones, between angles that are square, and angles that are not quite square, between equal and unequal spacing. This does not happen because the buildings are inaccurate. It happens because they are more accurate.*
>
> *The similarity of parts occurs because the forces which create the parts are always more or less the same. But the slight roughness or unevenness among these similarities, come from the fact that forces are never exactly the same. . . .*
>
> *And it is marked, in feeling, by a sharpness and a freedom and a sleepiness which happens everywhere when men and women are free in their hearts. . . .*
>
> *It is not necessarily complicated. It is not necessarily simple. . . .*

> *It comes simply from the fact that every part is whole in its own right.*
>
> *Imagine a prefabricated window which sits in a hole in a wall. It is a one, a unit; but it can be lifted directly out from the wall. This is both literally true, and true in feeling. Literally, you can lift the window out without doing damage to the fabric of the wall. And, in your imagination, the window can be removed without disturbing the fabric of what surrounds it.*
>
> *Compare this with another window. Imagine a pair of columns outside the window, forming a part of the window space. They create an ambiguous space which is part of the outside, and yet also part of the window. Imagine splayed reveals, which help to form the window, and yet, also, with the light reflected off them, shining in the room, they are also part of the room. And imagine a window seat leaning against the window sill, but a seat whose back is indistinguishable from the window sill, because it is continuous.*
>
> *This window cannot be lifted out. It is one with the patterns which surround it; it is both distinct itself, and also part of them. The boundaries between things are less marked; they overlap with other boundaries in such a way that the continuity of the world, at this particular place, is greater. . . .*
>
> *The timeless character of buildings is as much a part of nature as the character of rivers, trees, hills, flames, and stars.*
>
> *Each class of phenomena in nature has its own characteristic morphology. Stars have their character; rivers have their character; oceans have their character; mountains have their character; forests have theirs; trees, flowers, insects, all have theirs. And when buildings are made properly, and true to all the forces in them, then they too will always have their own specific character. This is the character created by the timeless way.*
>
> *It is the physical embodiment, in towns and buildings, of the quality without a name.* (Alexander 1979)

Notice the possibly disturbing implications for software if we are to take Alexander at face value: In an Alexandrian system there is never any sense of modularity at any level of detail. Our definition of abstraction (“Abstraction Descant”) allows for variations in the abstractions, but systematic ones. Is this definition flexible enough to capture the sense of freedom we get from Alexander’s descriptions? For example, in his description of a window, we sense that every part of the abstraction—the panes, the frame, the relation to parts not specifically part of the window—is a little different in each instance depending on the forces at work at the window site. Instances of classes can acquire some of this flavor by varying the state of each object, each one different according to its role in the system; but is it enough?

Windows are windows because of their behavior vis-à-vis people: You can look through them, open them for air and spring smells; they provide light, and sometimes they act like seats or meeting places. Certainly this seems like the sort of variation we can expect from an object system—to be able to define a class of object very abstractly according to its behavior—but are we, perhaps, expecting too much in the variations? For instance, some windows provide light and air, but others act as chairs if they have sills so constructed. And windows can be ambiguous in their surroundings—can we possibly accommodate this in a modern, structured software design? Perhaps; perhaps if we drop some of our expectations regarding reuse, for instance, and universal applicability of abstractions. I'll say more about this later.

Alexander felt that people did not understand the geometrical nature of the quality, but he is partly to blame. It is sometimes hard to extract the importance of geometry from *Timeless Way* and *Pattern Language* possibly because he does not want people to think that modern architecture, with its obvious simplified geometry—large rectangular buildings designed to be looked at and not lived in—is an example of good architecture. He laments:

> *I've known from various readings that the book has had, that most people do not fully understand that chapter [Chapter 26]. It's just too short and it does not fully explain itself—although I was aware that in that book I just could not do the topic justice. In other words, I became increasingly aware of the fact that my own understanding of this, among other things, existed at a very highly developed geometrical level and that all of what* The Timeless Way of Building (1979) *was about—all of its human psychological and social content, and all of its political and constructional content—could actually be* seen *in the geometry. That is, there was a particular morphological character that exists in buildings which have this quality and which does not exist in buildings which lack it—and furthermore, that this geometrical character is describable and quite clear. But although I knew that to be so, and thought that I had written about it, I actually had not. I thought that Chapter 8—which has to do with the morphological processes in nature—together with the patterns, and together with Chapter 26, must make this clear. But in fact they do not.* (Grabow 1983)

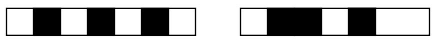
Alexander was surprised that people didn't understand that the geometrical nature of his pattern language was fundamental. He felt he always knew that the geometry had to be right, and he always knew that he could tell quickly that it was right or wrong. I don't think that even after Mexicali he fully appreciated the importance of geometry, certainly not to the extent he seems to in his study of Turkish carpets. The problem in Mexicali wasn't obviously that everything was fine except for a lack of appreciation for geometry—it was that and also that the general skill level was low and that families participating in the design were unfamiliar with the language and process. But from his experience Alexander began to believe that maybe there was more to the geometry than even he appreciated and that perhaps it was something the could be codified. Alexander said:

> *The point is that I was aware of some sort of field of stuff—some geometrical stuff—which I had actually had a growing knowledge of for years and years, had thought that I had written about or explained, and realized that, although I knew a great deal about it, I had never really written it down. . . .*
>
> *In a diagnostic sense, I can say that if this geometrical field is not present in something then there is something wrong there and I can assess that fact within a few seconds.* (Grabow 1983)

What is this geometrical field of stuff? We can understand some of it from two things he did: a series of experiments with black-and-white serial patterns and an analysis of his Turkish rug collection.

I'll start with the series of experiments because it predates the Turkish rug studies, although the results of the experiments make more sense in the context of understanding the rugs.

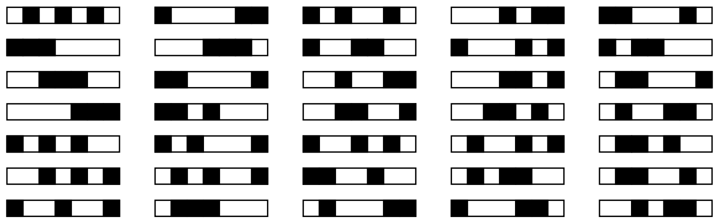
In the 1960s at Harvard, Alexander performed an experiment in which he asked subjects to rank-order a set of strips from most coherent and simple to least coherent and simple. There were 35 strips, each consisting of three black squares and four white ones. Here are two examples:



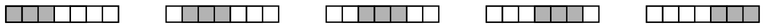
If there was good agreement among the subjects about which were simpler and more coherent, then there would probably be some truth to the conjecture that there is an objective quality of wholeness or coherence or simplicity. It turned out there was excellent agreement; as Alexander wrote:

> *First, Huggins [Alexander's coinvestigator] and I established that the relative coherence of the different patterns—operationally defined as ease of perception—was an objective quality, that varied little from person to person. In other words, the perceived coherence is not an idiosyncratic subjective thing, seen differently by different people. It is seen roughly the same by everyone.* (Alexander 1993)

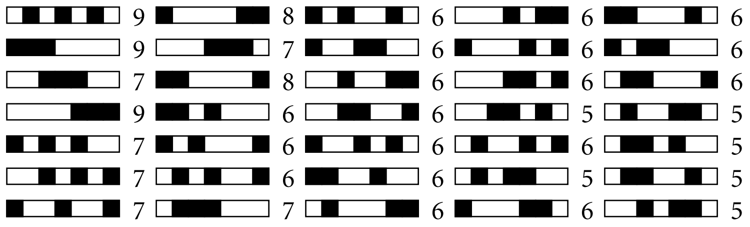
At the top of the next page are the 35 strips ordered according to the test subjects from most coherent at the top left, moving down the first column and then down each column from left to right, with the least coherent at the bottom right.



Next Alexander asked whether there was some way to explain this ranking. Oddly—I think—there is a way. It has to do with counting what Alexander calls *subsymmetries*. A subsymmetry is a symmetrical subsegment. Coherence is strongly correlated with the number of subsymmetries. Here's how you calculate this number: Consider all the subsegments of sizes 2, 3, 4, 5, 6, and 7. There is 1 subsegment of length 7, 2 of length 6, 3 of length 5, and so forth—21 in all. Here are the subsegments of length 3:



Now we simply count the number of subsegments that are symmetric. A subsegment is symmetric if it looks the same reversed. Let me repeat the previous figure with the number of subsymmetries to the right of each strip:



That there is such a correlation is remarkable. I can't say that I'm convinced that the ordering made by Alexander's subjects seems right to me; for example, I would have put the strip at the bottom of the first column ahead of the ones just above it. However, let's assume the data is accurate and try to understand the idea of subsymmetries.

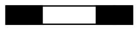
It seems that people prefer symmetries that appear on all levels—so in Alexander's formula, small symmetries count as much as large ones. The first strip (upper left) has symmetries all over the place, and it is pleasant to look at, but is it really more coherent than the one at the bottom of the first column? It seems more regular in some ways, and it is certainly more interesting—it seems more complex, but it seems easier to remember, since there is a simple alternation rather than the 1-2-1-2-1 of the other. Maybe it's more coherent. The idea of subsymmetries seems to take this into account, perhaps a little too simplistically. Alexander wrote:

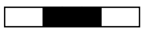
> *Thus, apparently. the perceived coherence of the different patterns depends almost entirely on the number of symmetrical segments which they contain. Since each of the segments which is symmetrical is a local symmetry I summarize this whole result, by saying that the most coherent patterns are the ones which contain the largest number of local symmetries or "subsymmetries."* (Alexander 1993)

When we look at his carpets, we'll see the same thing.

I spent a couple of hours playing with some variations on his computation and found one that correlated slightly better with experimental data and which fits his remarks on rugs as well. My computation adds additional bias in favor of doubled outer boundaries that diminishes exponentially as the boundary grows smaller.

For example, a strip like this has a bias added to its subsymmetry count (it is not a legal strip in his experiment):



This serves to boost the score of the strip that is third from the top in the first column and the one at the bottom of that same column. But the improvement is marginal. I suspect that perfect correlation requires a general algorithm like Alexander's, along with special cases for strips of overwhelming power. For example, it's hard to get the third strip from the top in the first column () to come out third best when in general subsymmetries or something like them have to count in order to get the simple alternation strip to come out first—in Alexander's subsymmetry algorithm it comes out as low as ninth, and in my revised one it comes out fifth.

Notice that although there is an algorithmic way to determine whether a strip is coherent (to some degree of accuracy), there is not yet a formative rule for it, though I suspect there would be, perhaps a context-free grammar. But Alexander didn't find one.

When he moved into a study of Turkish carpets, he moved into a vastly more complex world. Next we'll look at his surprising look into the carpets, and, even more surprisingly, we'll see some glimmer of connection to software.

❧ ❧ ❧

Alexander's foray into rugs follows the strip-research vein, but on a far less scientific basis. There are no studies, there are no numbers (though he does talk about the number of "centers" in a carpet correlating with its "wholeness"). And in general the results there are a little less compelling as argument but much more compelling in beauty.

In the early 1970s Alexander began buying Turkish carpets, religious Turkish carpets. He said:

> *I was extremely innocent when I started out. I simply liked them. My main concern was actually in their color. I was completely absorbed by the question of color but never thought it would have any serious connection to my work. Also, I never thought of my interest in these rugs as having to do with geometry.* (Grabow 1983)

He spent a lot of money—even getting into financial trouble; he became a rug dealer for a while—and he became known to Bay Area rug collectors. In fact, his carpets were once shown in a special exhibition at the DeYoung Museum in San Francisco.

Most people who collect rugs have a special interest, such the village where the rugs were woven or the treatment of a particular theme, but Alexander's rugs weren't in such neat categories—they were chosen because they had something special about them. Because he had so little money compared with the cost of each carpet, he spent a lot of time looking at them before he bought them. He wasn't especially aware of the special quality that set some carpets apart—even though his interest in carpets began in the midst of his quest for the quality without a name. His friends mentioned to him that his carpets had some special something and he said:

> *When people started telling me this I began to look more carefully to discover that there was indeed something I was attracted to in a half-conscious way. It seemed to me that the rugs I tended to buy exuded or captured an incredible amount of power which I did not understand but which I obviously recognized.*
>
> *In the course of buying so many rugs I made a number of discoveries. First, I discovered that you could not tell if a rug had this special property—a spiritual quality—until you had been with it for about a week. . . . So, as a short cut, I began to be aware that there were certain* geometrical properties *that were predictors of this spiritual property. In other words, I made the shocking discovery that you could actually look at the rug in a sort of superficial way and just see if it had certain geometrical properties, and if it did, you could be almost certain that it had this spiritual property as well.*
> (Grabow 1983)

Alexander taught some courses on the geometry of his rugs and discovered that the carpets shared this magical property with religious buildings and religious art as well. Over the next 20 years he prepared a book about it, recently published, called *A Foreshadowing of 21st Century Art: The Color and Geometry of Very Early Turkish Carpets* (1993). This is a remarkable book. It's about 9 x 12 inches and excellently bound. The paper is the sort you find in high-quality art-print books, and the reason is that it is full of gorgeous reproductions of his carpet collection.

The sad thing—for you—of course is that I cannot possibly reproduce any of the art from this book in such as way as to do it justice, so some of my comments will necessarily be a little on the abstract side (those of you who believe wholeheartedly in the power of abstraction to solve all the world's problem will have no trouble with this at all). However, I will try my hand at reproducing some of his examples. But for the real impact—and the beauty of his stunning carpets—buy or borrow the book and enjoy.

In this book Alexander says some pretty darn unbelievable things. First is that the beauty of a structure—a building, for instance—comes from the fine detail at an almost microscopic level. He takes the grain size for noticeable structure from that used in the carpets: one eighth inch. He wrote:

> *In short, the small structure, the detailed organization of matter—controls the macroscopic level at a way that architects have hardly dreamed of.*
>
> *But twentieth century art has been very bad at handling this level. We have become used to a "conceptual" approach to building, in which like cardboard, large superficial slabs of concrete, or glass, or painted sheetrock or plywood create very abstract forms at the big level. But they have no soul, because they have no fine structure at all. . . .*
>
> *It means, directly, that if we hope to make buildings in which the rooms and building feel harmonious—we too, must make sure that the structure is correct down to ⅛th of an inch. Any structure which is more gross, and which leaves this last eighth of an inch, rough, or uncalculated, or inharmonious—will inevitably be crude.*
> (Alexander 1993)

Second is that color and feeling also comes from this fine structure:

> *The geometric micro-organization which I have described leads directly to the glowing color which we find in carpets. It is this achievement of color which makes the carpet have the intense 'being' character that leads us to the soul.* (Alexander 1993)

Alexander feels that artists of the past—often of the distant past but as recently as Matisse, Bonnard, and Derain—had a better hold on beauty and that it is the task of late twentieth century artists to try to recapture the knowledge that seemed, perhaps, so obvious to these earlier artists as to be intuitive. Carpets provide a way to study this mastery because they are pure design, pure ornament, and their construction is so completely unconstrained by the materials of their construction as to allow the artist's true mastery to come forward. Alexander:

> *In a carpet, we have something which deals almost entirely with pattern, ornament. There is really nothing else: just the geometry and the color of the plane. As I began to enjoy carpets, I realized that the earliest carpets, especially, deal with this problem with enormous sophistication. The design of the carpet is essentially made of tiny knots—each knot usually about an ⅛ of an inch by an ⅛ of an inch. Each knot is a separate bit of wool, and may be any color, without any reference to the underlying warps and wefts. So it is a pure design, made of tiny elements, and which the structure (the design structure, the pure organization of the geometrical arrangement) is the main thing which is going on.* (Alexander 1993)

This is just a bitmap, but, as we'll see, with perhaps quite a number of color bits per pixel.

What is fascinating about this book is that Alexander is not afraid to come out and say that the power of these carpets comes at least in part from the need of these early artists to portray their religious feelings and needs. In fact, Alexander boldly tells us in the first paragraph of Chapter 1:

> *A carpet is a picture of God. That is the essential fact, fundamental to the people who produced the carpets, and fundamental to any proper understanding of these carpets. . . .*
>
> *The Sufis, who wove most of these carpets, tried to reach union with God. And, in doing it, in contemplating this God, the carpet actually tries, itself, to be a picture of the all seeing everlasting stuff. We may also call it the infinite domain or pearl-stuff.*
> (Alexander 1993)

The color of the carpets is paramount, and Alexander points out that usually the oldest carpets—even though they are the most faded—have the brightest and most brilliant colors. He says this is partly because the master dyer was an equal, in the twelfth century, to the master weaver. Such a master dyer served a 15-year apprenticeship, after which the apprentice was required to produce a color no one had seen before. Only after that could the apprentice become a master. Alexander notes that this training is the equivalent today of training as a theoretical physicist followed by training as a brain surgeon. Perhaps a dyer did not learn as much as someone would for either of these two professions, but it certainly says a lot about the importance of the dyer to Turkish society.

The depth of feeling in a carpet is related to a concept Alexander calls *wholeness*:

> *Both the animal-being which comes to life in a carpet, and the inner light of its color, depend directly on the extent to which the carpet achieves wholeness in its geometry. The greatest carpets—the ones*

> *which are most valuable, most profound—are, quite simply, the carpets which achieve the greatest degree of this wholeness within themselves.* (Alexander 1993)

Alexander proposes that wholeness is an objective concept—it has nothing to do with preferences or subjective feelings about the object that might display it. Like the strips we saw earlier, Alexander gives the reader a test to prove it, in which he shows us two pairs of carpets and asks an unusual question:

> *If you had to choose one of these two carpets, as a picture of your own self, then which one of the two carpets would you choose? . . .*
>
> *In case you find it hard to ask the question, let me clarify by asking you to choose the one* which seems better able to represent your whole being, the essence of yourself, good and bad, all that is human in you. (Alexander 1993, emphasis in original)

We are presented with two pairs of carpets. The first pair is the Berlin prayer rug and the Kazak from the Tschebull collection; the second pair is the Flowered Carpet with Giant Central Medallion and the Waving Border Carpet, both from Alexander's collection.

Alexander claims that almost everyone will pick the same carpet from each pair, because an objective something, a wholeness or oneness, comes through.

Up to this point in reading the book I thought maybe Alexander had gone a little off his nut and I was ready to quit, so I took the test to prove he was getting a little too fanciful at this stage of his career. I stared at the two carpets in the first pair and spent some time trying to second-guess him. Then I kept looking at the first rug until I felt as calm as I could looking at it. After, I looked at the second one, again until I became the most peaceful I could feel. Then I asked myself the questions Alexander posed, and I chose the one that made me more at ease while thinking about myself. I chose the Berlin prayer rug. This was the choice he predicted.

At this point I felt nervous.

He pointed out that the Berlin prayer rug is very well known, and sometimes people have seen it and perhaps don't remember it but unconsciously choose it because it is familiar. He then invites us to try it on the previously unpublished pair from his own collection: the Flowered Carpet with Giant Central Medallion on the left and the Waving Border Carpet on the right.

I looked at the Waving Border Carpet—the right-hand one—which has a large central octagon with four surrounding smaller octagons. They are mostly blue with intricate interlocking patterns inside. The large octagon has some red, white, and beige subpatterns. All the octagons are in a field of red. At the ends of the central part of the carpet are triangular pyramidal indentations—sort of partial shapes pressing into the center portion containing the octagons. These are darker, a deep blue-black—they are like an awakening into the center.

The primary feature of the carpet, though, is the waving border made of a degenerating vine. It is not very symmetric when you look closely at it, and it contains a recurring hook motif. Each bend of the vine contains a flower—maybe a tulip form—or a goddesslike figure—very abstract but distinct. The effect is that the motion of the border is so demanding that it enhances the calmness at the center of the carpet. I was immediately attracted to it.

But then I started to wonder how I was being gamed by Alexander—what was he trying to do? I looked at the other carpet—the left-hand one. It is partly destroyed—the upper right quarter and half of the very central portion are completely missing. The border is partly chewed away; the colors seem faded. The border is clearly less dramatic than the Waving Border Carpet. But the central part of the rug has spandrels of a color I had never seen before, a lace interlock of a deep green-black, but shimmering on a bed of slightly dark red. Sometimes the green comes to the fore, sometimes the red. The interlace is symmetric only at the grossest level, with almost every detail that should be the same, actually different. Each of the three existing spandrels are different from the others but seem of a kind. I found I was drawn to the upper-left spandrel, which is denser, more involved—a complexity that I followed and focused on for 10 minutes without noticing the time. I would be pulled in then pulled out, and as I moved my eyes from one spandrel to another, I crossed the central medallion, a simple blue, yellow, and red flower motif bordered in an Escher-field of abstract animal figures.

But it was clear that the first rug was the more profound and impressive to a trained eye, although I really wanted to look at the second one longer. Foo on you, Alexander, I said, you and your idiosyncratic punctuation and crazy theories—I'm choosing this lesser rug—the left-hand one—and this will prove that your damn ideas that I've puzzled over for years are worthless.

> *I believe that almost everyone, after careful thought, will choose the left-hand example. Even though the two are of roughly equal importance, and of comparable age, I believe most people will conclude that the left-hand one is more profound: that one feels more calm looking at it; that one could look at it, day after day, for more years, that it fills one more successfully, with a calm and peaceful feeling. All this is what I mean by saying that, objectively, the left-hand carpet is the greater—and the more whole, of the two.* (Alexander 1993)

My domestic partner made the same choices, feeling as I did that the left-hand carpet of the second pair showed significant flaws that mirrored something in her, though it was obviously the lesser carpet in some formal sense. (Later that week nine of 10 people I quizzed made the same choices.)

I kept reading.

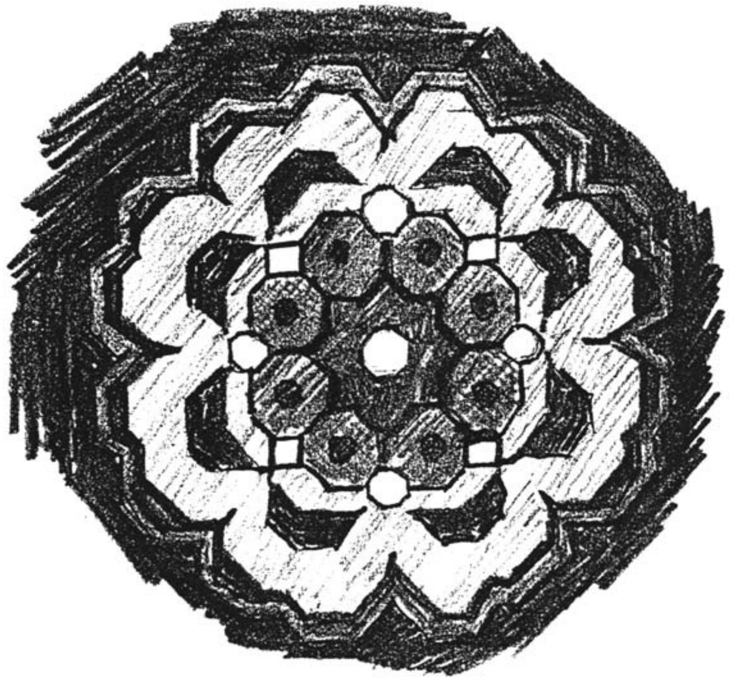
The book is in four parts: a theoretical discussion of where the quality of wholeness comes from, a discussion of dating the carpets and how they fit into a progression, pictures and descriptions of the carpets themselves, and finally a comparison of two carpets, the first at the start of the progression and the second at the end—using them Alexander shows us how far the art has degenerated. I want to talk only about the origins of wholeness.

A carpet is whole to the degree that it has a thorough structure of *centers*. Alexander starts us out with this definition:

> *As a first approximation, a "center" may be defined as a psychological entity which is perceived as a whole, and which creates the feeling of a center, in the visual field.* (Alexander 1993)

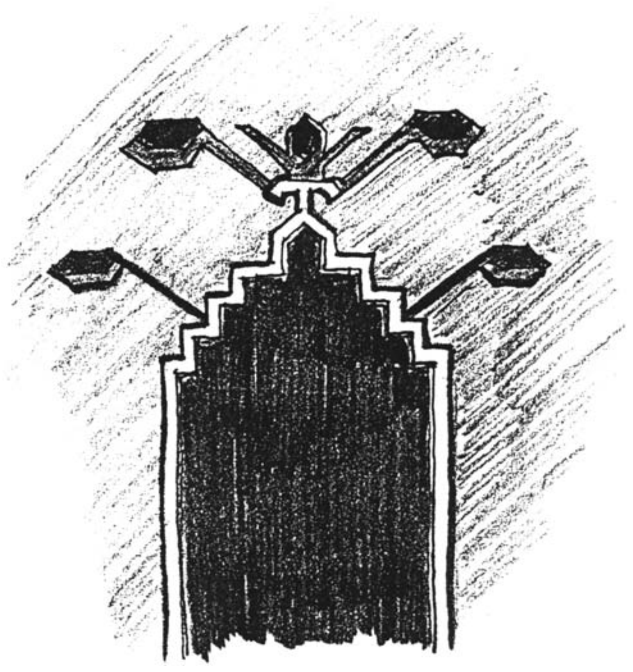
This definition doesn't help much, so he gives us some examples.

The following is an example of a center from a round blossom from the Blossom fragment:



Notice that this figure has a strong center—in the very middle. But that's not the main point. Each of the lighter octagons and diamonds forms another center, the darker dots at the centers of the smaller blossoms form others. The asymmetrical black leaves are kinds of centers. The sharp indentations of the outer press toward the middle, reinforcing the center. The Blossom center gives the impression that centers are all like mandalas—somewhat circular and concentric. Not so.

Here is a sketch from the niche of the Coupled column prayer rug:



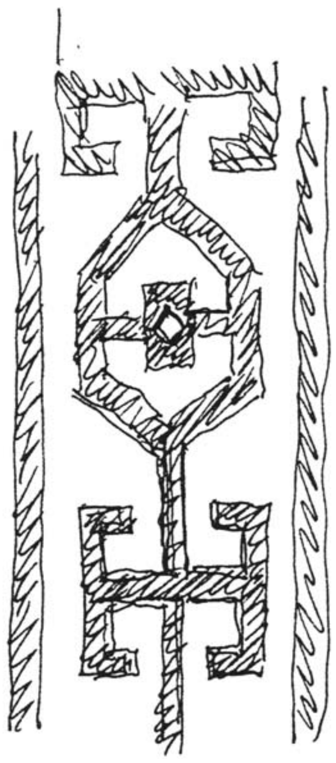
This form is a center because of the hexagonal archway, the steps, the arms with hexagons at the ends, and the lily at the top—not because of the overall shape.

The next point Alexander makes is that for wholeness, it isn't that a carpet has a single center but that it has a multiplicity of centers that reinforce one another by influence and by their structure, and that's what makes the difference.

Alexander says:

> *The degree of wholeness which a carpet achieves is directly correlated to the number of centers which it contains. The more centers it has in it, the more powerful and deep its degree of wholeness.*
> (Alexander 1993)

In fact, the more centers there are and the more intertwined and interlaced they are, the more whole the carpet becomes because the centers are denser. At the top of the next page is an example of this from the border of the Seljuk prayer carpet. Both the dark design elements and the lighter background form centers wherever there is a convex spot, wherever linear parts cross, and at bends. There are perhaps a dozen or more centers here.



Centers are made up of local symmetries.

> *1. Most centers are symmetrical. This means they have at least one bilateral symmetry.*
> *2. Even when centers are* asymmetrical*, they are always composed of smaller elements or centers which* are *symmetrical.*
> *3. All centers are made of many internal local symmetries, which produce smaller centers within the larger center (most of them not on the main axis of the larger center), and* have a very high internal density of local symmetries. *It is this property which gives them their power.* (Alexander 1993)

One of the interesting things about the carpets is that the sense of symmetry remains even when the parts that are supposed to correspond are not exactly alike. At the top of the next page is an interesting example. Look at it quickly and decide whether you think it is symmetric. Looking closely at it, it's clear that it is crudely symmetric, but this is enough for most people to see it as very symmetric. There is actually something remarkable at work here: It is not just that the crudeness of this form—both in the original and in my even cruder reproduction—does not get in the way of its beauty; in fact, it enhances that beauty. On the wall above my writing place I have a photograph of a Greek "shack" taken by

Barbara Cordes. The shack is made of stone with a red-tile roof. Each stone is of a different size—some roughly cut, others seemingly randomly selected. The roof tiles are all different colors and roughly arranged. There is a deep blue frame window, and the shack sits behind a dead bush. The reason I have the picture there is that it is relaxing and helps me write. Its beauty, somehow, comes from the irregularity of the construction. Here is what Alexander wrote about a similar building, the famous House of Tiles in Mexico City:

> *We have become used to almost fanatical precision in the construction of buildings. Tile work, for instance, must be perfectly aligned, perfectly square, every tile perfectly cut, and the whole thing accurate on a grid to a tolerance of a sixteenth of an inch. But our tilework is dead and ugly, without soul.*
>
> *In this Mexican house the tiles are roughly cut, the wall is not perfectly plumb, and the tiles don't even line up properly. Sometimes one tile is as much as half an inch behind the next one in the vertical plane.*
>
> *And why? Is it because these Mexican craftsmen didn't know how to do precise work? I don't think so. I believe they simply knew what is important and what is not, and they took good care to pay attention only to what is important: to the color, the design, the feeling of one tile and its relationship to the next—the important things that create the harmony and feeling of the wall. The plumb and the alignment can be quite rough without making any difference, so they didn't bother to spend too much effort on these things.* They spent their effort in the way that made the most difference. *And so they produced this wonderful quality, this harmony . . . simply because* that is what they paid attention to, and what they tried to produce.
> (Alexander 1991, emphasis in original)

The reason that American craftsmen cannot achieve the same thing is that they are concerned with perfection and plumb, and it is not possible to concentrate on two things at the same time—perfection and the field of centers.

> *In our time, many of us have been taught to strive for an insane perfection that means nothing. To get wholeness, you must try instead to strive for* this *kind of perfection, where things that don't matter are left rough and unimportant, and the things that really matter are given deep attention. This is a perfection that seems imperfect. But it is a far deeper thing.* (Alexander 1991, emphasis in original)

At this point it is clear that Alexander's earlier work on subsymmetries plays right into the wholeness of rugs: The strips with more subsymmetries were seen as more coherent than those with fewer. A subsymmetry is like a center, and the more centers the more whole the carpet is. Furthermore, subsymmetries range from the size of the entire strip down to the smallest place where symmetry is possible—pairs of squares. The same is true of centers: They must exist at all levels of scale. In fact, carpets with all the other characteristics for wholeness but lacking centers at all levels will not achieve wholeness and so can be boring. Alexander states the rule thus:

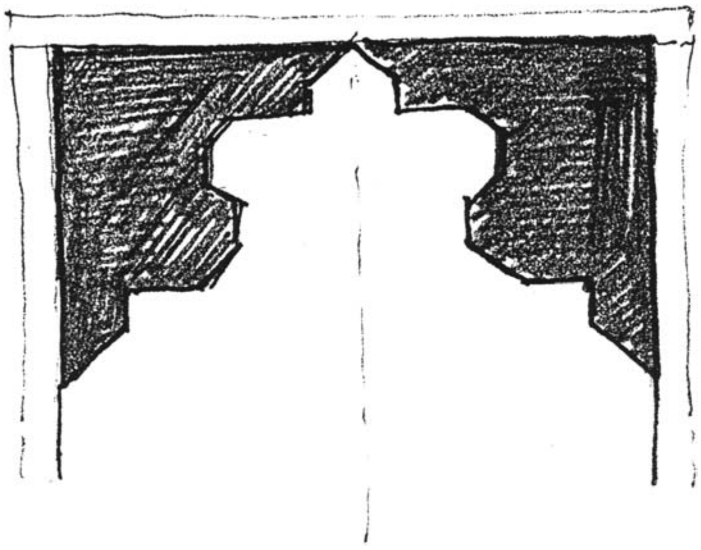
> *A center will become distinct, and strong, only when it contains, within itself, another center, also strong, and no less than half its own size.* (Alexander 1993)

This is true not only for the same reasons it is true in the strips but also because without large structure, the design cannot hold together—it becomes merely a jumble of isolated design elements, each of which might hold attention but without the large glue it will not be calm and whole—it would be like gluing together several masterpiece paintings at the edges. With this definition it is apparent that the goal of a carpet is to present a single center—the one at the largest scale—which necessarily must be constructed from other centers that support it.

One of the things I noticed about the Flowered Carpet with Giant Central Medallion was that the interlaced green and red spandrels held my attention. This is an example of Alexander's notion of the strong use of *positive space*. In such use both the positive and negative spaces have good shape and form centers. In this way the density of centers can be higher, and the degree of wholeness stronger. This concept should be familiar because it is like those Escher drawings of fish, for example, in which both the figure and ground can be seen as fish. It is a little more subtle in this niche, which appears at the top of the next page.

The shapes of the white convexities are strong as are the black convexities that are white concavities. The figure is bilaterally symmetric and leads us to strong centers at the half-hexagon at the top and in the spandrels.

Now that we have centers—lots of them—we need to make them distinct—the centers need to be differentiated; recall that in Alexander's *A Timeless Way* (1979),



Alexander talked at length about differentiation. At the top of the next page is a very rough sketch of the central star of the star Ushak rug.

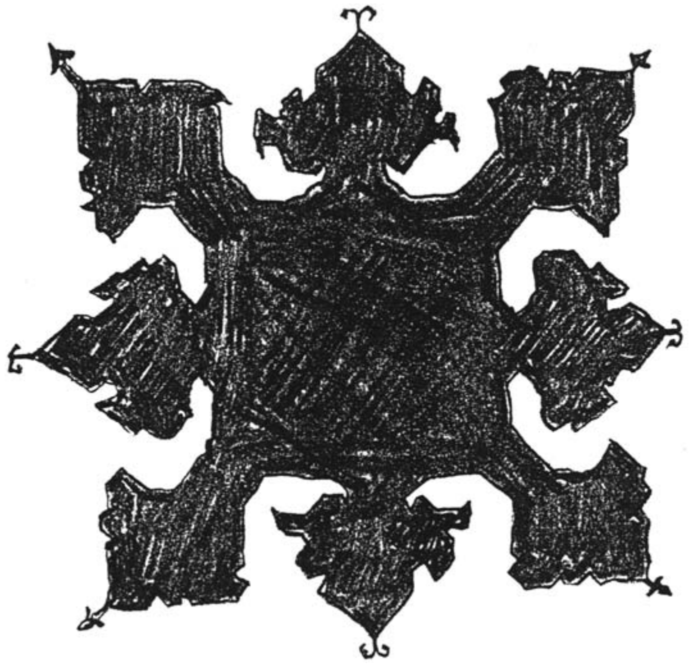
Alexander says the star achieves its distinctiveness from five sources:

> *1. The centers next to the figure—those created by the space around it—are also very strong.*
>
> *2. These strong centers are extremely* different *in character from the star itself—thus the distinctness is achieved, in part, by the differences between the centers of the figure, and the centers of the ground.*
>
> *3. There are very strong color differences between field and ground.*
>
> *4. The complex character of the boundary line seems, at least in this case, to contribute to the distinctiveness of the form. . . .*
>
> *5. The hierarchy of levels of scale in the centers also help create the effect, by increasing the degree to which the form is perceived as a whole, entity, or being in its own right.* (Alexander 1993, emphasis in original)

The definition of a center now is:

> *Every successful center is made of a center surrounded by a boundary which is itself made of centers.* (Alexander 1993)

That this is a partially circular definition seems to startle Alexander, although we computer scientists and mathematicians find it simple. It reminds me of the defi-



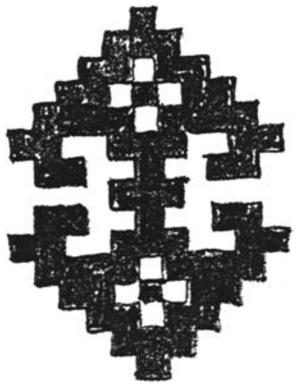
nition of *system*, which I'll talk about once we're through with Alexander's wild ride.

Alexander goes back to the bead game and talks about the difficulty of finding two-dimensional patterns with high degrees of subsymmetries like the strips we saw earlier. He points out one that is seen in a couple of rare Konya carpets—it is a figure that is almost immediately familiar and unique (top of the next page).

There clearly are many centers here, and the figure ground distinction is amazing. There are embedded crosses, giving the entire figure the feeling of a being, and there are almost smaller beings within the larger one.

This figure lies in an 11 x 15 grid. It is one shape out of 10^{47} possibilities. Because of this, Alexander is enthralled by the creative process require to find it. He says something interesting about this process, which we can liken to the design process in software:

> *[T]he greatest structures, the greatest centers, are created not within the framework of a standard pattern—no matter how dense the structures it contains—but in a more spontaneous frame of mind, in which the centers lead to other centers, and the structure evolves, almost of its own accord, under completely autonomous or spontaneous circumstances. Under these circumstances* the design is not thought out, conceived—it springs into existence, almost more spontaneously, during the process by which it is made.



And, of course, this *process corresponds more closely to the conditions under which a carpet is actually woven—since working, row by row, knot by knot, and having to create the design as it goes along, without ever seeing the whole, until the carpet itself is actually finished—this condition, which would seem to place such constraint and difficulty on the act of creation—is* in fact *just that circumstance in which the spontaneous, unconscious knowledge of the maker is most easily released from the domination of thought—and thus allows itself most easily to create the deepest centers of all.*
(Alexander 1993, emphasis added)

The implication is clear: When we carefully design or work from a standard pattern, we do not achieve artifacts of the deepest meaning and wholeness. Perhaps the results are pleasant, but they are eventually boring and funky—perhaps they are commonplace or do not serve their purpose as well as they could. Alexander says he has evidence for his bold statement—he leaves it to the next book (not yet published) in the sequence, *The Nature of Order*—but there is not enough room in such a small book as *A Foreshadowing* to go into it.

This is also the way the best (creative) writing is done. When you sit down at your writing place, sometimes you are transformed into a mere scribe as the power of the story or poem takes over—it is an adventure to see where it will lead. Many writers have a ritual or set of superstitions that they hope will lead them to this magic place where the story becomes guide. When asked why they write, some writers say it is to find out what happens in the end.

I write poetry and fiction as well as essays. In all three, but especially in poetry, I am often shocked to read what I wrote, because it rarely corresponds to any plan I've made. Books on writing—at least the most recent ones—tend to discourage overplanning the story or poem. The question becomes—and you must answer this for yourself—does this clearly creative process have a place in a software process?

The last topic Alexander addresses is one I casually brought up just a bit ago—the emergence of beings.

I'll leave it to Alexander to introduce the topic:

> *I now present the culmination of the argument. This hinges on an extraordinary phenomenon—closely connected to the nature of wholeness—and fundamental to the character of great Turkish carpet art. It may be explained in a single sentence:* As a carpet begins to be a center *(and thus to contain the densely packed structure of centers. . .)*, then, gradually, the carpet as a whole also begins to take on the nature of "being." *We may also say that it begins to be a picture of a human soul.*
>
> *The subject is delicate, because it is not quite clear how to discuss it—not even how to evaluate it—nor even in what field or category to place it. It opens the door to something we can only call "spirit" and to the empirical fact—a fact of psychology if of nothing else—that after all, when a carpet does achieve some greatness, the greatness it achieves seems to lie in the realm of the spirit, not merely in the realm of art.* (Alexander 1993, emphasis in original)

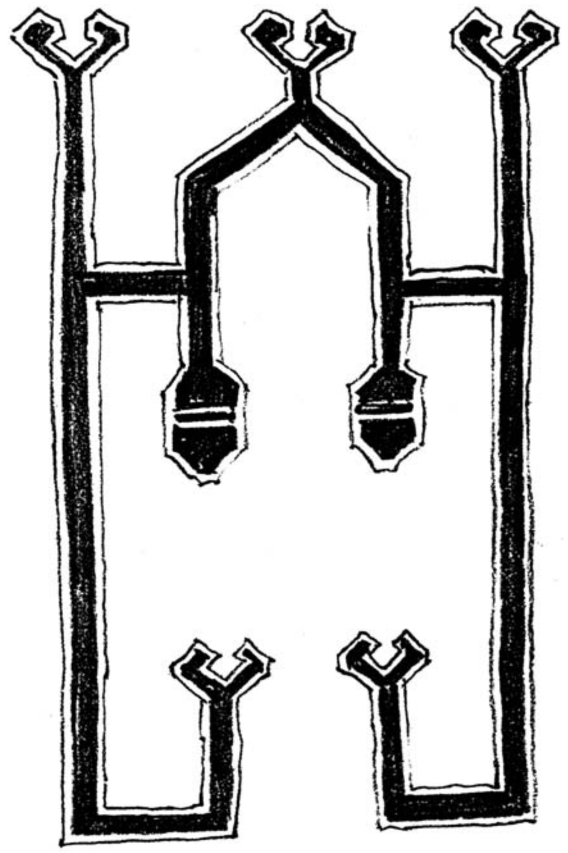
What is a being? It's a powerful center that transcends the merely fascinating. It is a center that takes over and reflects the human and perhaps god-spirit. It is autonomous, a "creation unto itself." The figure at the top of the next page is the being that appears in the Seljuk prayer carpet.

I think it's rather impressive.

Alexander ends his book by telling us:

> *I see the beginnings of an attitude in which the structure may be understood, concretely, and with a tough mind—not only with an emotional heart. And I see the rebirth of an attitude about the world, perhaps based on new views of ethics, truth, ecology, which will give us a proper ground-stuff for the mental attitude from which these works can spring.*
>
> *I do not believe that these works—the works of the 21st century—will resemble the Turkish carpets in any literal sense. But I believe some form of the same primitive force, the same knowledge of structure, and the same desire to make a work in which the work carries and illuminates the spirit—will be present.*
>
> *I am almost certain, that in the 21st century, this ground-stuff will appear.* (Alexander 1993)

Perhaps.



Well, what can this possibly have to do with software? Maybe not too much, but Alexander's definition of center sounds familiar to me. Here is a definition of system I've been using:

> *A* system *is a set of* communicating components *that work together to provide a comprehensive set of capabilities.*

Of course, a component is another system. The nature of a system is such that at almost any granularity it looks the same—it is a system. This is easy: A system is composed of subsystems that achieve certain effects by exhibiting behavior in response to requests. The system and each subsystem are defined behaviorally according to the roles and responsibilities of that system or subsystem. Each subsystem can be viewed as a center, and the near circularity of Alexander's definition of center is reflected here: The goal is to build a system which is necessarily made up of other systems.

We can compare this with the definition of *autopoietic system*, which is a term from a field that studies systems from the point of view of biology—you could call it a field that has a mechanistic view of biological systems.

> *An* autopoietic system *is organized as a network of processes of production of components that produces the components that: (1) through their interactions and transformations continuously regenerate and realize the network of processes that produced them; and (2) constitute it as a concrete entity in the space in which they exist by specifying the topological domain of its realization as a network.* (Varela 1979)

If you read this definition carefully, you will find that it isn't so very different from my definition of "system" or Alexander's definition of "center". When we put together an object-oriented thing, it is a system, not a program. The difference between a program and a system is precisely the characteristic of a system having many centers or ways of approaching it—from the vantage point of any subsystem, the rest of the system is a server—whereas in a program, you generally have a single way of viewing it, usually from the top down.

The most wonderful thing can happen if you construct a system that can be customized or specialized—it becomes a *framework*.

> *A* framework *is a system that can be customized, specialized, or extended to provide more specific, more appropriate, or slightly different capabilities*

If you have a framework, you can use it for different purposes without having to recode it all. It is in this sense—and in this sense only—that objects and object technology provide reuse. There is no such thing as (easy) reuse in a general sense in which you can stockpile components except for the most trivial and inconsequential.

Building a system is like making a Turkish carpet: It requires an enormous amount of work, concentration, inspiration, creativity, and, in my opinion, it is best done piecemeal exactly the way the best Turkish master makers did it—by starting with an idea and partly designing it and then seeing where it goes.

Only in some systems does the being emerge, the framework that can be used and reused which gives systems and objects their spirit.

Like the best Turkish carpets the best systems are unique, but the work will not be wasted if you can continue to use it for new but related purposes. It is an asset and an inspiration to others who follow to try to do as well. A system can afford to be different—and perhaps over time we will develop mechanisms in our overly speed-conscious programming languages and system-building languages to make it easier to produce highly differentiated and center-full systems—I have hopes for the twenty-first century too.

What are patterns, then? Perhaps they are for Alexander what they are by analogy for computer scientists: just a way to remember what we've forgotten about buildings so that when we set out to build towns and buildings, cottages and homes, paths and places to congregate, we don't forget the stuff we need to help create the centers, to give them life, give them the quality without a name, the being that emerges at last.