RESTORING FUNCTION AND FORM TO PATTERNS

DCI AS AGILE'S EXPRESSION OF ARCHITECTURE

James O. Coplien

Email cope@gertrudandcope.com

Abstract

More than 15 years ago, we laid the foundations for the software pattern discipline — a discipline that has become an enduring staple of software architecture. The initial community distinguished patterns from other approaches with high ideals related to systems thinking and beauty. Many of these ideals reflect age-old thinking about systems and form from fields as diverse as urban planning and biology. However, the community's body of literature soon degenerated to the point of showing little distinction from its contemporaries except in its volume. This paper is based on a talk that described a vision for systems thinking in IT architecture, drawing on the principles that originally launched the pattern discipline. The talk argued that by focusing on system form, including the form of function, we can build better systems with lower lifetime cost than is currently possible with patterns. Together we'll consider both popular and emerging architectural notions to ground this vision.

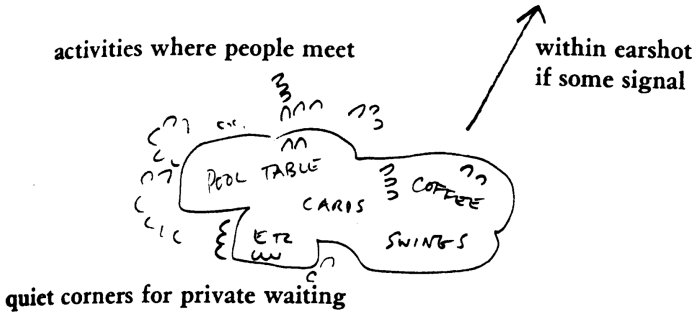
Patterns' Origins in Architecture and in Alexander's Ideals

Patterns have been popularized by the architect Christopher Alexander in the 20th century as a design formalism. Alexander uses patterns as a guide to construction at human levels of scale, from towns and neighborhoods down to houses and rooms. Patterns are system elements grounded in experience that compose with each other in a process of piecemeal growth and adaptation:

> We may regard a pattern as an empirically grounded imperative which states the preconditions for healthy individual and social life in a community. (Christopher Alexander, *The Oregon Experiment*, Ch. 4)

Every pattern is grounded in experience, has form, and is human-focused. For example, consider the pattern A PLACE TO WAIT:

> *The process of waiting has inherent conflicts in it.*
>
> One on hand, whatever people are waiting for— the doctor, an airplane, a business appointment— has built-in uncertainties, which make it inevitable that they must spend a long time hanging around, waiting, doing nothing.
>
> On the other hand, they cannot usually afford to enjoy this time. Because it is unpredictable, they must hang at the very door. Since they never know exactly when their turn will come, they cannot even take a stroll or sit outside...
>
> …
>
> Therefore:
>
> *In places where people end up waiting, create a situation which makes the waiting positive. Fuse the waiting with some other activity—newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simply waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence.* (Christopher Alexander, *A Pattern Language*, 1977, pp. 707—711)



Alexander relates three properties of patterns that have become staples of software development. Patterns go hand-in-hand with an underlying process. First, the process of building with patterns is incremental:

> Piecemeal growth is based on the assumption that adaptation between buildings and their **users** is necessarily a slow and continuous business which cannot, under any circumstances, be achieved in a single leap. (Christopher Alexander, *The Oregon Experiment*, p. 69)

Second, it is grounded deep in human sensibilities, more so than in just engineering practices:

> *What we need is a way of understanding the forces which cuts through this intellectual difficulty and goes closer to the empirical core.*
>
> . . .
>
> *To do this, we must rely on **feelings** more than intellect.* (Christopher Alexander, *The Timeless Way of Building*, Chapter 15)

Third, the process is iterative:

> ... several acts of building, each one done to repair and magnify the product of the previous acts, will slowly generate a larger and more complex **whole** than any single act can generate (Christopher Alexander, *op. cit.*, Ch. 24)

The notion of *process* is fundamental to patterns' place in system evolution. We grow a system one pattern at a time. While patterns are elements of form, it is also useful to think of them as functional transformations on a system that map the system from one state to another. So the pattern LIGHT ON TWO SIDES OF EVERY ROOM transforms a dark house into one where every room is bathed in light from multiple natural sources. We apply this pattern, at least conceptually, to a house that already has found its building site and has been roughly partitioned into rooms.

The house at any stage is a result of a history of pattern compositions. We cannot throw together a house with a random sequence of patterns: order matters. All of these constraints, taken together with all of the patterns, define a partial ordering of pattern application. The patterns are like elements of a language, and the constraints define the language. Indeed, the total system of patterns is called a *pattern language*. This notion of language is fundamental to patterns and their place in systems thinking. This is why Alexander talks about "several acts of building," each one of which is "done to repair and magnify the product of the previous acts." No pattern stands alone. It is always part of a system, a Whole. The Whole is a gestalt that arises from the combination of its pattern parts, yet it is greater than the sum of the parts.

These attributes of the pattern process foresee many modern software practices. However, though the software community embraced Alexander's patterns in the 1990s, few of these ideals show up in contemporary software pattern practice, but rather in other facets of modern software construction. In particular, many of these values are prominent in Agile software development.

Patterns, in general, are neither about the built world in particular nor about software in particular. Alexander's works describe a deeper and more universal philosophy of design that transcends disciplines. This is to be expected from someone who played a prominent role in the Design Movement of the 1980s — a dialog among the great design thinkers in numerous disciplines at the time. But Alexander might object and claim that these insights go even deeper than just design: that they touch on the very nature of form, of matter, and of being in their own right. Patterns encode, at human scale, the processes and forms by which natural processes unfold in everyday systems. These processes add texture and structure to undifferentiated matter. For example, when an egg develops from a zygote to a blastosphere, cell reproduction is initially homogeneous. Then cells begin to specialize, and an embryo forms with differentiated tissue. Most natural processes fit this metaphor. Alexander postu-

lates that the emergence of a city plan or housing architecture follows the same deep processes. The question is: does this universality of design also apply to software?

The morning that I gave this keynote, I received a wonderful mail from Gordon Globius, who is Professor Emeritus of Psychiatry and Philosophy at UC-Irvine. He is studying semiotics: the role of signs in language. He points out that Saussure, who founded semiology, tells us that:

> Psychologically our thought — apart from its expression in words — is only a shapeless and indistinct mass. ... the somewhat mysterious fact ... that language works out its units while taking shape between two shapeless masses ... concepts are purely differential and defined not by their positive content but negatively by their relations with the other terms of the system.Their most precise characteristic is in being what the others are not. (Course in General Lingustics)

This sounds very similar to a concept from physics called *spontaneous symmetry breaking*, which is the process by which matter is transformed in a way that actually decreases its level of entropy. It increases structure. This phenomenon is sometimes called self-organization. Alexander's houses are, in some sense, self-organizing, if one includes the house dweller as an active part of the evolving system.

Alexander starts by describing the importance of geometry in design, and he articulates it using the formalism of symmetry:

> ... we shall see that almost everything about life in buildings can, in the end, be understood through symmetries, and that indeed, there may be a way in which the concept of wholeness, and the field of centers, when understood dynamically, can be understood completely in terms of sequential unfolding of symmetries. (Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life,* 2002, p. 242)

However, he continues and cautions against a naive dependence on simple symmetry:

> Living things, though often symmetrical, rarely have perfect symmetry. Indeed perfect symmetry is often a mark of death in things, rather than life. I believe the lack of clarity in the subject has arisen because of a failure to distinguish overall symmetry from local symmetries. (Ibid., p. 186)

and:

> In general, a large symmetry of the simplified neoclassicist type rarely contributes to the life of a thing, because in any complex whole in the world, there are nearly always complex, asymmetrical forces at work—matters of location, and context, and function—which require that symmetry be broken. (Ibid., p. 187)

Globius' focus on language offers the tantalizing perspective that this same symmetry breaking that underlies architecture also applies to language and the nature of human reasoning. It may define the way that brains (and minds) reason about form — the fundamental pillar of architecture. It may also define that way that end users conceptualize software programs through an interactive interface.

The rest of this paper is a reflection on fundamental aspects of those deeper principles, and on what they ultimately portend for software design today.

The Software Community Ideals

Patterns gained a foothold in the software world when the nascent Hillside Group met in 1993 to explore the application of Alexander's ideas to software. The next year they met again in California. The group challenged itself to look forward

a decade and to describe how patterns had changed the world. The following hopes and dreams were in their thoughts (James O. Coplien, personal notes from the meeting):

- Patterns have brought to life that software is art, bifurcating the industry into pseudo-scientists and artisans
- They have brought a new programming language, and have enriched the vocabulary used by others.
- Patterns' major use is in neural networks, where they are used to describe high-level firing patterns and distribution of computation in emerging highly distributed systems.
- Patterns have caused three major universities to shift software from their engineering and science programs, into liberal arts, with concomitant changes in curriculum (art history is now a prerequisite for a comp sci degree)

The pattern community was inspired in large part of a vision of pattern languages: collection of patterns that formed a system. Earlier discussions about patterns at OOPSLA and at an IBM workshop in Thornwood, New York had revolved in part around the early thesis work of Eric Gamma, who was documented micro-architectures in descriptions that would come to be called patterns. The core of the early pattern community, which called itself the Hillside Group, distinguished "generative patterns" from so-called "Gamma patterns" that lacked this linguistic generativity.

The pattern community knew that it was breaking new ground and decided to take a conservative tact. This conservatism was reflected in several unwritten mores of the culture: we would keep things simple and direct instead of "going meta;" we would be inclusive to ensure that we explored the concept in a way that built on broad contributions rather than presuming that any single group of people had a monopoly on what constituted a pattern. However, we would ensure that the members of the original inner core were materially engaged in each pattern that went forward, using a process called shepherding, which has expanded into many other conferences' reviewing culture.

The community sought its footing for a couple of years, and then the number of attendees, and conferences, and patterns, grew. People started experimenting, as engineers are wont to do, and this experimentation led to small steps away from the core foundations. Many of these foundations were lost over time. If we consider Alexander's notion that patterns should focus on adaptation between the product and the end user, it is difficult to find it as a foundation even of the early mainstream patterns. The noteworthy exceptions were the patterns from an offshoot community in HCI, which held dearly to this principle, but that effort was more of an HCI-focused effort than a pattern-focused effort. Alexander's notion that patterns should be incremental and iterative was also lost by the mainstream pattern community, largely because the grander notion of pattern languages never took root. In terms of relying on feeling more than on intellect, this never took root in the patterns themselves, though the pattern community clearly emphasized its identity as a community of people in relationship over any technical consideration. In short, the pattern community was left with little more than a convenient public outlet for possibly good ideas. But they never came close to having the morally grounded imperative of any community, in accordance with the Alexandrian vision. (James Coplien, *The Culture of Patterns*, http://docs.google.com/a/gertrudandcope.com/viewer?a=v&pid=sites&srcid=Z2VydHJ1ZGFuZGNvcGUuY29tfGluZm98Z3g6ZDg1YWFjMThjZDBkZjQ2)

One challenge of patterns is that they were *too* fluffy. It was hard for consultants to sell patterns. Patterns sold books, not consulting — and not even that many books. Academics quickly discounted them as unsuitable for research publication, since they were not original. Since the fabric of the pattern community largely comprised consultants and academics, patterns lost mindshare among key spokespersons in the industry. Patterns coasted forward as a grass-roots movement that came to realize only a glimmer of Alexander's ideals.

Reformulating the Ideals

Few, if any, of Alexander's original key ideas survived through the to the software pattern community. Among the noteworthy exceptions are the sense of community behind the adoption of patterns, and the strong sense of a human element among the participants. But this human element seldom shines through in the pattern literature per se.

These pattern values nonetheless found a new outlet in Agile software development. Agile found stronger business footings than patterns ever did, playing to the hopes of management for business flexibility and to the hopes of developers for increased autonomy. It is not only true that the key initial proponents of Agile had been key players in the pattern community, but that the key elements of the pattern value system became explicit in Agile as well. We can return to the key ideas of focusing on users, wholes, and feeling.

ALEXANDER	SOFTWARE
Adaptation between software and its users	People (users): Mental models
Slowly generate larger and more complex wholes	Form: Object-orientation and domain analysis (wholes)
Rely on feeling more than intellect	Function: The form of function (feeling)

People and Mental Models

The adaptation between a system and its **users** is mostly about the users. In architecture this came to be called user-centered design. Software had long had advocates for human-centered software. Much of this advocacy was rooted in the object-oriented programming discipline. A bit of history is instructive in understanding how this came to be.

While object orientation had its origins in Simula 67, the term actually was coined by Alan Kay. Alan describes object orientation like this:

> In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computer all hooked together by a very fast network. (Alan Kay, *The Computer revolution hasn't happened yet*, http://www.thedavincipursuit.net/page24/page25/files/kay.pdf)

Form: Objects and Wholes

The software community has long appreciated the importance of form. Almost ever since Fred Brooks at IBM stated using architecture as a metaphor for programming-in-the-large in software, the phrase "software architecture" has been an industry staple. We are concerned with the differentiation of the "**whole**" into parts that can be built, independently and simply, and then composed into a system.

The object paradigm has arisen as a popular style to describe system parts. As we'll explore in detail later, "object-oriented" is really a misnomer in light of the fact that most programmers actually build classes rather than objects.

One would think that because of this focus on form in objects, and because of its manifestation in the user interfaces of interactive programs, that this value would have a place of honor in Agile thinking. For historic reasons, it does not. Agile tended to take an anti-knowledge, pro-skill outlook on development. There is a kind of passive-aggressive anti-

intellectualism in Agile that works against the kind of expertise that offers first-hand insight into form. Agile instead takes the position that form emerges through experimentation over time, using feedback to re-shape the system.

This, however, would be a temporary situation. While some of the foundational Agile thinkers still cling to the notion of YAGNI ("you ain't gonna need it") for system form, the deeper thinkers have returned to a tradition of up-front thinking. This perspective has returned in a healthy way that eschews much of the heavyweight documentation and premature code mass that usually went hand-in-hand with the previous generation's software architecture. But the essentials of system form remain. Bob Martin tells us,

> One of the more insidious and persistent myths of agile development is that up-front architecture and design are bad; that you should never spend time up front making architectural decisions. That instead you should evolve your architecture and design from nothing, one test-case at a time.
>
> Pardon me, but that's Horse Shit. (Robert C. Martin, World Expert on The Scatology of Agile Architecture, http://blog.objectmentor.com/articles/2009/04/25/the-scatology-of-agile-architecture)

Function: Does it have form?

Hofstader, in his book *Le Ton Beau de Mareau*, describes the difficulty of translating between languages. Programmers are faced with this very problem when translating between an internal human language and the many languages of computing. If we are to reflect human mental models in a programming language, what is a good translation? Hofstader concludes that a good poem is one that elicits the same **feeling** in the reader as for the original feelings of the poet. Function isn't just about correctness, but exhibits itself deeply in a sense of satisfaction, joy, or at least catharsis in the users of our software.

The Big Picture

We find these values crystallized in the Agile Manifesto. The Manifesto expresses adaptation of a system through its reflections on "Individuals and interactions" as well as "Responding to change." That's the focus on **users**.

The Manifesto reflects on the place of **wholes** in its reflections on documentation: that the whole is better than the documentation. A good architecture supports the discovery process and the ability of the programmer to understand the code, which is a key foundation of working software. C. A. R. Hoare emphasizes the importance of code comprehension:

> There are two ways of constructing software. One is to make it so simple that there obviously are no deficiencies. The other is to make it so complicated that there are no obvious deficiencies... The price of reliability is the pursuit of the utmost simplicity.

ALEXANDER	SOFTWARE	AGILE
Adaptation between software and its users	People (users): Mental models	Individuals and interactions over processes and tools
Slowly generate larger and more complex wholes	Form: Object-orientation and domain analysis (wholes)	Working software over comprehensive documentation
Rely on feeling more than intellect	Function: The form of function (feeling)	Responding to change over following a plan.

The Manifesto also raises the question of human **feeling**: We want end users to be content and perhaps even happy. They are content if their expectations are met. As we work to align our expectations of a product with theirs, both we and the

system go through many changes. It is crucial to respond to change. Discomfort comes from forcing someone into an overly inflexible plan.

These are well-known provisions of the Agile Manifesto:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> Kent Beck
> Mike Beedle
> Arie van Bennekun
> Alistair Cockburn
> Ward Cunningham
> Martin Fowler
> James Grenning
> Jim Highsmith
> Andrew Hunt
> Ron Jeffries
> Jon Kern
> Brian Marick
> Robert Cecil Martin
> Steve Mellor
> Ken Schwaber
> Jeff Sutherland
> Dave Thomas
>
>

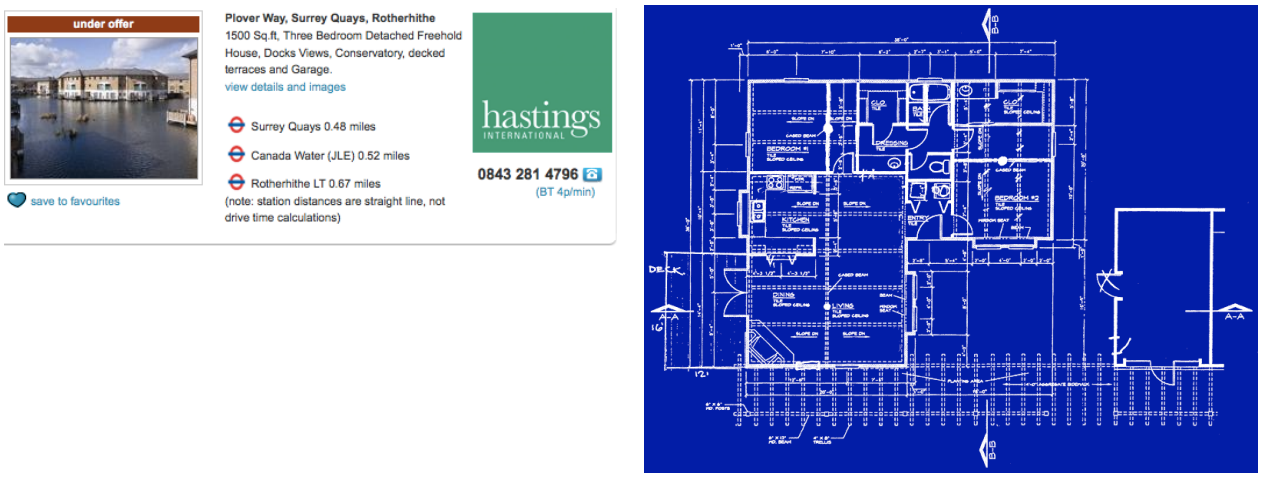
Most people interpret the Manifesto in terms of its provision for development team members. They are the individuals who interact. It is their job to build working software. It is them who shouldn't over-plan. They should be flexible with the software. But the team is a small part of the overall puzzle of delivering software: in the end, it is the software, and not the team, which stands at the center. We want the software to interact with end users as individuals. We want the software to be in accordance with the end user mental model instead of some instruction manual. We want the customer to feel engaged, as Brenda Laurel describes attendance at a theatre production as a metaphor for software enactment (Brenda Laurel, *Computers as Theatre*). That means that software is a service — not a product. And we want the software to follow end-user whims both in the short term (to be able to support the user's newfound need by anticipating scenarios well) and in the long term (by offering the flexibility for new features).

Abandonment of People and Human Mental Models

Almost all programming has its roots in mental models. While object-orientation trumpets these foundations, FORTRAN was there first with its claim to representing the mathematical FORmulæ in the minds of its mathematically-minded programmers, and COBOL was the first to dare speak of comprehensibility of code by managers and business people. Modern programming seems to have lost this focus, favoring an engineering view (think girders and beams) over the human view (think external elevation). The industry has even dared justify this perspective with the moniker "Software Engineering," as though it had roots in a real science instead of something called Computer Science.

Object orientation came to value this engineering perspective more and more in the 1980s with its focus on coupling and cohesion, the depth of inheritance hierarchies, and unit testing. More and more programming concerns moved further and further from the end user. Many software engineering measures and concerns became rooted in pseudo-science rather than the mental models either of the programmer or the end user.

The pattern community stood staunch in its insistence on visual representations of design. Alexander had always insisted on these because of his focus on geometry, and the place of geometry in human comprehension of comfort, habitability, convenience, and of most aspects of design that conveyed value. However, the software community would move more and more to engineering representations of form, instead of forms rooted in human mental models. Notations such as UML became gratuitous adornments to the mental model. There was a broad interest in formalizing patterns.



The customer view and engineering view of a house

Engineering diagrams have a use, but Christopher Alexander has warned against them as inflexible master plans that do not reflect the sensibilities of the end user. I like to show people the interface of my iPhone while on a trip to another city. I captured it from my wife's phone and I reproduce it below. Can you tell what city I was in when I captured the screen?

What happens when you count on engineering approaches for testing

If you're observant you notice the 20:57 time at the top, and note that the only clock exhibiting that time corresponds to Boston. So that's a good guess, and it turns out to be right. All of the digital clocks are wrong. The program is probably right — right in the sense that the green bar comes up for all the tests. But as Jef Raskin advises, the interface is the pro-

gram. The code is just the un-lean stuff that has to go along with the interface to make it work. It's the interface that has to work.

Interfaces are what we design, develop, and debug — or we should. Too often, the interface is an afterthought. We build the system bottom-up, starting with the database and adding the application logic, and then hiring out the design of the interface later. And we test the interface last — at which point changing it would require a major change to most of the software architecture.

But, today, we've lost the link between the human mental model and the program. The problem shows up in many ways. We work on classes instead of objects. Classes are structure rather than form, and they have only a vague correspondence to the end-user mental model. They usually end following the programmer mental model, and it's usually the end user who suffers. I'm sure that the person who wrote the code for inserting pictures in word processor paragraphs ran the test sand the green bar came up just fine every time. That I can't get the machine's mental model to conform to my notion of where the picture belongs somehow eluded the tests.

Back to Form

Early object-oriented programmers understood the link between the interface and the program. Brenda Laurel talks at length about the relationship between the worlds of the programmer and the end user. Smalltalk programmers, particularly those using VisualWorks, lived in an intensively visual world. The worlds of the programmer and end user are inextricably linked in working software. More so, it is the end user model that usually must prevail, though more generally good development strives to align the expectations of the end user and the team. This is a key tenet of Lean: to align expectations by bringing the entire team together to reason both about the process and the form.

Geometry and Architecture

The architecture of the built world is about form. We can interpret this perspective in two ways. In one sense, form is the essence of the building structure: it is what is left if you take all the specific structure away. Viewed another way, architects focus on the form *between* the built structures. It is an architect's job to carve out space. Walls, floors and roofs just delineate the space: the space itself is the important result.

Alexander was deeply inspired by the Tao te Ching — a classic work of Chinese philosophical literature. We find these principles of form even in those roots. For example,

> Thirty spokes meet at a nave;
> Because of the hole we may use the wheel.
> Clay is moulded into a vessel;
> Because of the hollow we may use the cup.
> Walls are built around a hearth;
> Because of the doors we may use the house.
> Thus tools come from what exists,
> But use from what does not.

Compare that quote with this pattern from Alexander:

> POSITIVE OUTDOOR SPACE
>
> Outdoor spaces which are merely "left over" between buildings will, in general, not be used...
>
> Make all the outdoor spaces which surround and lie between your buildings positive. Give each one some degree of enclosure; surround each space with wings of buildings, trees, hedges, fences, arcades, and trel-

> lised walks, until it becomes an entity with a positive quality and does not spill out indefinitely around corners.

Such space arises from differentiation of the larger original space — a kind of local breaking of symmetry.

Christopher Alexander reminds us, again, that geometry drives much deeper into our understanding of things than we give it credit for.

> A pulsating, fluid, but nonetheless definite entity swims in your mind's eye. It is a geometrical image, it is far more than the knowledge of the problem; it is the knowledge of the problem, coupled with the knowledge of the kinds of geometrics which will solve the problem, and coupled with the feeling which is created by that kind of geometry solving that problem. (Christopher Alexander, *The Timeless Way of Building*, Chapter 9)

What's more, is that we gain a hint here is that this geometry is dynamic. It is "pulsating" and "fluid." It is clearly about form. And we see the word "feeling" appear again for the first time in a long while. It is what Alexander calls a Whole, and it solves a problem — whether physical, economical, or psychological — for some person. Or for the universe.

If we get form right, we provide comfort to our end users. A good clock interface won't have the opportunity to confuse the user. Perhaps it should display only analog images and omit the digital representations. At least there would be no opportunity for our earlier observed error in that kind of interface. But these are details. A good interface maps onto the end user mental model of their world.

The Form of Software

The debate occasionally arises of whether software has shape or form. The original goal of object orientation was to capture human mental models in the computer, so the computer could serve as an extension of human memory and processing power. The obvious challenge is the technology mismatch between wetware and software. It is important both that the end user be able to perceive and reason about the items in computer memory, and that the user be able to interact with those items.

We call those items *objects*. Each object that represents part of the human mental model pairs up with one or more *view objects* that carry the knowledge of how to visually represent a given object to a given end user. Each mental-model object may have multiple views for any single user: for example, a given user may explore a data set either as a pie chart or a bar graph. The system may cater to different users with different views, because every person is different, and each may have his or her own mental model of the world.

The computer display, then, displays a geometry of the end-user mental model. Each screenful is a system: a cogent reflection of the end-user mental model at any given time. Why does it work well? Geometry is one of the major organizing principles of human cognition. Our brains are highly optimized to pick out geometries in tight coupling with the visual cortex. For example, we instantly recognize straight horizontal and vertical lines and tend to to filter out those that are less straight or angular, and this selection takes place at a very low level of wetware processing. Each of these screens, these systems, might well be formalized as a pattern language. It reflects an "end state" of some sequence of patterns that has been applied again and again in the past and which will recur again in the future.

Last, we enable the end user to interact with these views. An object called the Controller creates and coordinates views. It also coordinates the display and the input device for operations such as selection. The combination of the human mental Models — represented as objects in computer memory — together with the Views and the Controller, form a pattern language. The language is called Model-View-Controller-User, or MVC-U.[1] Its patterns include the following (Trygve Reenskaug, *The Model View Controller: Its Past and Present*, 2003):

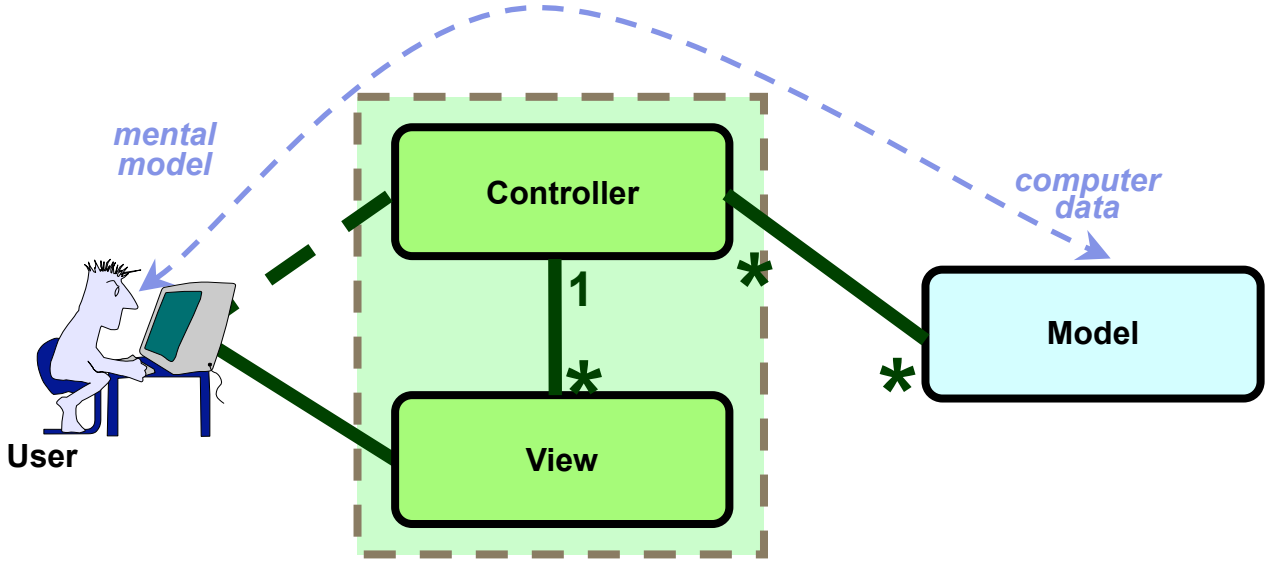
---

[1] We make the "U" explicit to emphasize the place of the User in the Whole built by the pattern language.

INTEGRATED DOMAIN SERVICES
LINE DEPARTMENT OWNS DOMAIN COMPONENTS
MENTAL OBJECT MODELS
PERSONAL INFORMATION SYSTEMS
DOMAIN/USER MATRIX
MODEL/EDITOR SEPARATION
INPUT/OUTPUT SEPARATION
TOOLS FOR TASKS
TOOL AS A COMPOSITE
SYNCHRONIZE SELECTION
SYNCHRONIZE MODEL AND VIEW

These patterns represent a geometry in the *programmer's* mental model. It is a geometry designed to support, in a general way, the structuring of the relationship between the wetware and the software. Programmers are people, too, and their mental models weave together with those of the end user in a program.

MVC gives the users to directly manipulate objects in computer memory as though they were real-world objects mirrored in their own mental models. So as I work with my word processor here, I feel as though I am writing on a paper or at least typing on a typewriter, and directly manipulating the model of the paragraph that exists in my mind. The Model contains *data* that encodes the *information* in my mind (there is no information in a computer; it comes into being only in the presence of human interpretation). The Controller and the View together form a *tool* that provides the illusion of direct manipulation to the end user. The job of the tool and of MVC overall is to align the geometries of the human mind and computer memory.



The result is the direct manipulation metaphor: the sense that the end user is directly manipulating the real world. The end user feels that they are in control. I am in control of the content and formatting of the paragraph I am currently working with. Sometimes the word processor gets in the way and I lose control — such as happens when I use another word process to try to put a picture right in the middle of a paragraph where I want it. The programmer's mental model of my world missed the mark. The word processor's architecture is wrong. It causes me to feel I am not in control. It causes me to be surprised by the program's response. In the extreme, it renders the program unusable.

Why do these lapses in usability arise so often in software? Too often, it seems, software engineers eagerly pursue the mastery of patterns of their own mental models instead of going back to the original end user mental models. It is somewhat like focusing on the scaffolding and according second-class status to the cathedral being constructed within the scaffolds. Many programmers view MVC as just a variant of the OBSERVER pattern (Eric Gamma et al., *Design Patterns: Elements of Re-Usable Object-Oriented Software*), but that is a naïve nerd-centric view that tends to discard the human element. And as we'll see later on, there are even more important end-user patterns beyond those honored by MVC-U.

Alexander reminds us that the essence of design is to leave the end user in control. Alexander believes that great homes are designed by their inhabitants, guided by the timeless pattern languages of their culture:

> On the other hand, people need a chance to identify with the part of the environment in which they live and work; they want some sense of ownership, some sense of territory. The most vital question about the various places in any community is always this: Do the people who use them own them psychologically? Do they feel that they can do with them as they wish; do they feel that the place is theirs; are they free to make the place their own? (Christopher Alexander, *The Oregon Experiment*, p. 38)

The Emperor's New Patterns

It doesn't take too much imagination to see that the software pattern discipline has missed the mark of human habitability. Software engineers have retreated into software engineering. Software architects have retreated into esoteric areas of notation and abstract principles such as reuse and interoperability. It isn't that their patterns are unimportant. They reflect human mental models. It's just that the primary human focus is on the programmers rather than on the end users. They've solved half the problem, that of the scaffolding. But even more disappointing, most software "patterns" fail to meet even the most basic criteria of being a pattern, and the industry has now arisen to a level of arrogant self-righteousness that attempts to defend, yet not properly justify, its ignorance. It might better be said that the industry is irrevocably apologizing for this lapse by citing the Vernacular Rule: meaning is defined by use rather than history. And they are right. So we leave the pattern people to their vernacular world, and we move on to the time-honored foundations that once were the hope of the nascent pattern community.

Is there hope? Do these ideals live on today? Strangely enough, some of them do. Patterns probably failed because they pushed the Open Source metaphor too far. The pattern community established an ethic of freely publishing patterns with the intent that they be re-copied and disseminated. The community took great pains to make sure that each pattern was self-standing, that was an enabling specification that could touch put the designer in touch with their inspiration to the degree that they could take it from there. It's difficult to sustain a consulting business around a body of literature that is designed to stand alone. Indeed, one of the founders of the software pattern discipline lamented in April of 1994 that "Fact is, my patterns effort have been cutting into revenue and it can't continue."

Nonetheless, the pattern ideals and foundations found a new outlet in about 1997: the market packaging called XP. And in 2001, seventeen males, almost all of them consultants, would come together in a three day meeting to write the Agile Manifesto you can find earlier in this paper.

ALEXANDER	AGILE MANIFESTO
The most vital question about the various places in any community is always this: Do the people who use them own them psychologically?	Individuals and interactions over processes and tools
Once the buildings are conceived like this, they can be built, directly, from a few simple marks made in the ground—again within a common language, but directly, and without the use of drawings.	Working software over comprehensive documentation
All decisions about what to build, and how to build it, will be in the hands of the users.	Customer collaboration over contract negotiation
Adaptation between buildings and their users is necessarily a slow and continuous business.	Responding to change over following a plan

Agile could have brought newfound hope to Alexander's vision. However, some of Alexander's key notions fell as casualties of the transition. One of them was the notion of geometry and *form*. Disillusioned with the structure-laden heavyweight architectures that were coming out of the UML world in the 1990s, XP and Agile declared: "You ain't gonna need it!", or "YAGNI", which was broadly interpreted as a call to stop doing up-front work. In fact, the culture gave rise to a new set of profanities such as BUFD ("big up-front design") in a buzzword-laden culture that struck fear into the hearts of managers who depended on the insights that their domain experts lent to the system form. It's better to use a process that allows programmers to craft system form a piece at a time than to be held hostage to a small number of individuals, such as domain experts or architects, who might leave your company any day or, worse, ask for a raise. Thus TDD receive broad support and displaced the timeless forms and pattern languages of the end-user world.

Such pattern thinking that survived focused on the mental model of the lowly, poorly-paid programmer rather than the high-risk, expensive architects. The *Design Patterns* book (Gamma et al., 1994) unwittingly led the way. It made life good for the consultants.

Back to the foundations of Architecture

This sounds like a crisis in software architecture, but in fact, the problem drives even much deeper. The software community viewed patterns as *static* elements of recurring *structure*, codified as *form*. That is, form was little more than a compressed representation of recurring structure. There was a lot of evidence to convict patterns of being just that. Alexander clearly underscores the importance of geometry in his writings and in his practice. When designing a community, Alexander lays out flags delineating the placement of buildings on the site. One can stand in each virtual building and assess the vista from windows-to-be. Below is picture of Alexander's layout of the Eishin school outside Tokyo.

However, architecture has a hidden side that doesn't show up in a casual view of form. Time is as important to architecture as structure is. This is nowhere more obvious than in Stewart Brandt's wonderful book *How Buildings Learn*. The

book is full of photographs of buildings, taken years apart, showing how buildings radically change their function over time while retaining much of their original form. Churches become restaurants; military barracks become research labs; horse stables become restaurants, offices, and a host of other human dwellings.

Yet *How Buildings Learn* makes time explicit and separate from form: a lens through which we assess the evolution of form. Time and form are in general much more subtly related. The epitome of this subtlety can be found in the Japanese concept of *ma*, which is usually translated "space-time." It includes the temporal synchronization of events in a certain spatial context.

The Function of Form

Alexander, whose work was heavily influenced by Oriental philosophy, tends towards this subtle confluence of form and time. His works long have been viewed as reflecting only the form of structure: the differentiation of space. However, consider the following quote:

> And finally, of course, I want to paint a picture which allows me to understand the patterns of events which keep on happening in the thing whose structure I seek. In other words, I hope to find a picture, or a structure, which will, in some rather obvious and simple sense, account for the outward properties, for the pattern of events of the thing which I am studying. (Christopher Alexander, *The Timeless Way of Building*, Chapter 5, 1979)

This is no late-breaking refinement of Alexander's ideas, but one of the fundamentals of his early work. Mull over the haunting phrase *patterns of events*. This phrase doesn't speak as much to form as it hints at placement of things within a space or the state of a space. What are the events of a kitchen? In an American farmhouse, they include cooking; welcoming guests (but only from the neighborhood), and having a winter afternoon coffee. It might also serve as the location for the children's evening homework projects. It is roughly the same long-term invariant form, but it is a form that serves and supports these transitions in state. These transitions tend to recur; there tend to be cycles of life that happens there. And the room supports those cycles — those patterns (Christopher Alexander, *The Timeless Way of Building*, Chapter 4):

> To understand this clearly, we must first recognize that what a town or building is, is governed, above all, by what is happening there.
>
> I mean this in the most general sense.
>
> Activities; events; forces; situations; lightning strikes; fish die; water flows; lovers quarrel; a cake burns; cats chase each other; a hummingbird sits outside my window; friends come by; my car breaks down; lovers' reunion; children born; grandparents go broke. . . .
>
> My life is made of episodes like this.
>
> The life of every person, animal, plant, creature, is made of similar episodes.
>
> The character of a place, then, is given to it by the episodes which happen there.
>
> *Those of us who are concerned with buildings tend to forget too easily that all the life and soul of a place, all of our experiences there, depend not simply on the physical environment, but on the patterns of events which we experience there.*

He continues:

> *Compare the power and importance of these events with the other purely geometrical aspects of the environment, which. architects concern themselves with.*

> Compare, for instance, two ways of including water in a building.
>
> Suppose, on the one hand, that there is a concrete reflecting pool outside your room with no purpose except to reflect the sky.
>
> And suppose, on the other hand, that there is a stream outside your room, with a small rowing boat on it, where you can go, to row, lie on the water, struggle against the stream, tip over . . . .
>
> *Which of these two makes the most difference to the building? The rowing boat of course, because it alters the entire experience of the building.*
>
> It is the action of these moments, the people involved in them, and the peculiar situations, which make the impression on our lives.
>
> The life of a house, or of a town, is not given to it, directly, by the shape of its buildings, or by the ornament and plan—it is given to them by the quality of the events and situations we encounter there. Always it is our situations which allow us to be what we are.
>
> It is the people around us, and the most common ways we have of meeting them, of being with them, it is, in short, the ways of being which exist in our world, that make it possible for us to be alive.
>
> *We know, then. that what matters in a building or a town is not its outward shape, its physical geometry alone, but the events that happen there.*
>
> . . .
>
> *A building or a town is given its character, essentially, by those events which keep on happening there most often.*
>
> . . .
>
> *And just the same is true in any person's individual life...*
>
> . . .
>
> *Of course, the standard patterns of events vary very much from person to person, and from culture to culture.*

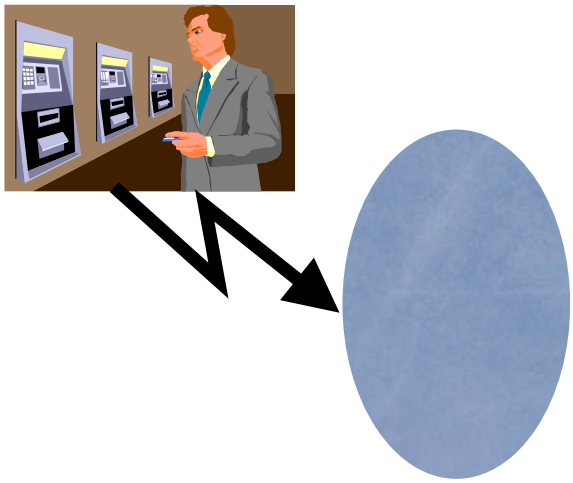
These images offer a glimmer of what time means to building architecture. Form does not follow function; form *has* function.

These thoughts invoke the roots of patterns in classical Chinese and Japanese thought. The Japanese have a word, *ma* (間) that is commonly translated *space-time*. These two dimensions, held to be orthogonal in Western thought, can be viewed as much more closely related in Japanese culture. This seems consistent with Alexander's notion of patterns of events.

What does it mean to software?

The Form *of* Function

Consider software for a cash machine. It is written in an object-oriented programming language, and the objects reflect a combination of the end user mental model and programmer mental model. We want to explore this notion of the "patterns of events" that happen there. We can run several experiments and instrument the code to help provide us answers.



Object method invocations must be the atomic building blocks of events, and any temporal patterns must be derived from their sequencing. So we can instrument each method of the program. First, we have each method print out its object ID when it is invoked. The log might look like this:

0145234428
0142366280
0283346250
0347212938
0324426294
0264274548
0374616738
0164571830
0173646284
0324426294
0145234420
0264274548

Well, not much of a pattern there. How about if we have each method print out its *class* every time it is executed? A class is a way to group objects by common structure. Because patterns are about form, there certainly must be patterns there:

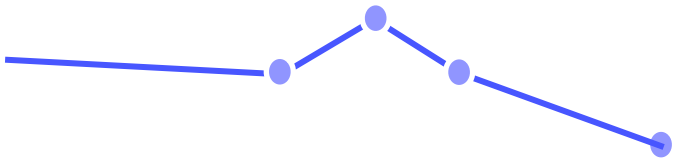
SavingsAccount
CheckingAccount
Euro
SavingsAccount
SavingsAccount
Krone
InvestmentAccount
SavingsAccount
Shekel
CheckingAccount
PhoneBill
Dollar

While there is a soft of rhythm, it is upset by the presence of the oddly uncharacteristic PhoneBill near the end of the list. We can make vague predictions about the nature of the next item in the sequence but it is not yet a true pattern.

On the other hand, we can have each method print out the *role* that its object is playing at the time the method is invoked. The role of an object is just the name of an object within a given use case:

SourceAccount
DestinationAccount

Amount
SourceAccount
DestinationAccount
Amount
SourceAccount
DestinationAccount
Amount
SourceAccount
DestinationAccount
Amount



This sequence represents an indisputable pattern. Yet it is a pattern in *time*, rather than a pattern in the usual sense of form. That is, function has form.

The DCI Architecture: Form of Function and Function of Form

The above exercise is an encouraging illustration about the patterns that exist in system function — a form that plays out in a role model of the business interactions. In the vein of Agile and its focus on "individuals and interactions," these are patterns in the space of human-computer dynamics.

Traditional object-orientation has missed this role perspective. Few modern object-oriented programming languages offer any way to express the form of function. This is curious, because one of the most instructive ways to view a programming language is in terms of its features for expressing form. Templates express generic forms; classes, object form; overloading, the form of behavior in different contexts; and so forth. Procedures, which capture the form of function, have long disappeared from prominence in the programming scene. What remains of them are methods or member functions that are finely splintered snippets of algorithm that are kept subordinate to the class structure. Even then, it is almost impossible to see how these snippets compose into orchestrations that have business value or that match the end-user mental model.

If we look carefully at end-user mental models, they feature both the form of the data and the form of the algorithm. We think about our savings accounts and checking accounts when we approach an ATM. We think about our flight (a reified event!) and our seat when we approach a plane reservation system. Classes represent such concepts well.

When our end users are ready to actually use the ATM, they bring another set of mental models into their minds. Ask yourself to write down the scenario of transferring money between two of your accounts in your bank. Be sure to keep the formulation general. You will probably come up with something like:

> The account holder selects the source account for the transfer, as well as the destination account and the amount to be transferred. The bank debits the amount from the source account and credits the amount to the destination account. The bank then notifies the account holder that the transfer has taken place.

This is obviously a use case written in free-form. The underlined phrases constitute *actors* in a use case. The use case formalism has long honored actors as a primary building block, and actors such as account holder and bank are familiar to us. Programmers usually view use cases as a way to delineate the boundary between the customer world and their own world. Yet here we see entities such as source account and destination account which, though part of the customer world, must be reckoned with in the programmer world. They, too, are actors. But from a linguistic perspective they are simply generic names for objects, each one of which collects a vastly wider collection of behaviors. These actors help us focus on the behaviors relevant to their respective scenarios. We can think of them as the names that the objects take on

for this use case. We may call your savings account the source account, and your checking account the destination account. These names are *roles*. For the purpose of this use case, the savings account plays the role of a source account and the checking account plays the role of a destination account.

This notion of role appears only rarely in design methods (one example is Trygve Reenskaug's OORAM), and even more rarely in programming languages. Though object orientation was supposed to capture the end user mental model, and though it did well to capture the form of data, it failed to capture the form of function. That form lives in the roles. Roles are the forgotten building block of software.

Historically, the main culprit has been the predominance of classes in programming languages. Most programmers program classes; very few of them program objects. Programming languages such as self (Chambers and David Ungar, *Self: The Power of Simplicity*, http://research.sun.com/self/papers/self-power.html) make it easier to reason about objects than when programming in Java or in C++. Modern languages such as Ruby, Python — and even Visual Basic — have basic features that speak of more to an object mindset than to a class mindset, though class-minded users can continue to remain stuck in their traditional thought pattern and programming style.

Yet, classes are here to stay. They offer software engineering advantages such as static type analysis, and they still form a significant part of programmers' mental models.

Time gets in the way

The fundamental problem is that software, like a building, is designed as a product, while in practice its utility is as a service. That is why Alexander repeatedly directs us to the importance of "patterns of events" in design. They are even more obviously prominent in software. Software has no value as a product: it has value only when running in a computer. As a product, it is the code of classes. As a service, it is running objects. Computers add instantiation and time to classes to give us objects.

An end user brings two mental faculties to a user interface: *thinking*, and *doing*. They first ponder what the objects of a system *are*: savings accounts, locations in the neighborhood, and restaurants and menus. Having sorted that out, they then sort out what the objects *do*: transfer money, find an optimal walking route, or make a reservation for dinner. Of course, at run time, both of these things are wrapped up in one kind of concept: an object. The object is both the locus of identity (what the system *is*) and of behavior (what the system *does*). It is crucial that we support this aspect of the end user mental model. If we don't, there is likely to be discord between what the program does and what the end user expects. Again, a good example is to try to place a graphic in the middle of a text paragraph using a common word processor.

Because these represent two mental faculties of real people, they also show up in two different mental faculties of programmers. Programmers are people, too. They typically use domain engineering, or a "find-the-nouns" exercise, or their experience, to identify the what-the-system-is aspect of the software. That is the part that is most often called software architecture and which we most often think of as system form. This form is, in fact, relatively independent of the system requirements! Why? Because in the end, software is a service rather than a product. What we sell is the service. Jef Raskin admonishes us that the interface *is* the program (Jef Raskin, *Humane Interfaces*). The code is just the crap that needs to go along with the interface to make it work. However, as we know, we *do* need this code. It is the classes. It is a substantial part of the programmer mental model, and it is a repository for the templates that produce the what-the-system-is part (the *thinking* part) of the end-user mental model.

The *doing* part of the end-user mental model lies in the use cases. This is also part of the end user mental model: it is what the system *does*. We showed above that this mental model also has form. It is the form of function, and it is expressed in the roles of the mental model.

These two aspects of the mental model come from quite different mental processes and can best be elicited using very different analysis and design techniques. While the what-the-system-is form evolves quite slowly, the what-the-system-does side evolves quite rapidly. We serve our markets by introducing ever new what-the-system-does stuff.

Unfortunately, classic object-oriented programming languages missed out on this concept of role. The only tool left to the programmer was the class. The class became the home both for what the system *is* and for what the system *does*. Both of these were often combined in a single interface. And who can blame the programmer for doing that? After all, the end user has only a single element in their mental model: the object. Objects combine both what-the-system is (data) and what-the-system-does (methods) in the end-user mental model, so why not combine those in classes? After all, objects come from classes.

The problem lies in the fact that behavior has a role structure that is difficult to reconcile with the class structure. The class structure is dead, static, and local. The real architecture of a program exists when the program exists: at run time. Its architecture lies latent in its objects and their behavior. Objects are *ma*. A program's architecture doesn't any more lie in its classes than the architecture of a house lies in its blueprint. The blueprint of Chartres captures neither its architectural grandeur nor the events of reverent prayer and worship there. We cannot examine the class structure at compile time and, at the same time, reason about run-time *system* behavior — the behavior that results from composing those methods at run time. Such reasoning is difficult because object-oriented design and programming slice and dice use cases across the what-the-system-is boundaries, rather than following the more natural role boundaries of the human mental model. Such reasoning is in fact in general impossible because programmers not only divide this functionality across class boundaries, but they refine this behavior in sub-typing hierarchies that can can be reasoned about only at run time. This leaves programs completely incomprehensible by the programmer. Object-oriented method dispatch is GOTO on steroids.

Enter testing

To succeed, then object-orientation needed to fall back on testing. What cannot be reasoned about in code authorship must be left to run time. The predominate myth is that we can improve software quality by independently designed models of code execution that we run in parallel with the code, controlling the code to run with certain constrained values and testing for the expected outputs. Of course, those models are also static.

Though we cannot reason about system behavior at run time, we can reason about the behavior of local methods. This has led to intense interest in unit testing and in driving architecture and development by those functions that you can reason about: hence, Test-Driven Development (TDD). This focus misses out on the emergent complexity that happens between the objects, as the "Thirty spokes hath the nave" image speaks to us from the Tao te Ching. It also completely misses out on the end-user mental models, both of what the *system* is and of what the *system* does.

Tony Hoare forewarned against this approach to development, and it is worth re-stating:

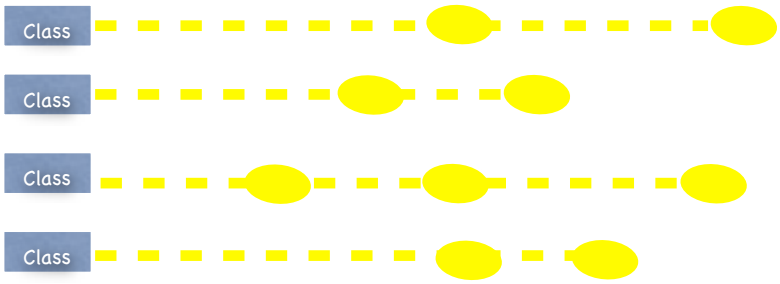
> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies.
> (C. A. R. Hoare)

The elements of form: What the system *is*, and what the system *does*

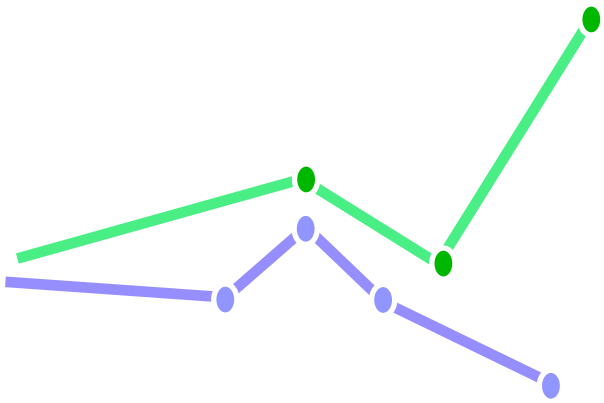
This brings us back to the fundamentals of architecture: capturing form. And it brings us back the fundamentals of Alexander's approach to design and his utmost concern for patterns of events.

As mentioned above, the real architecture of a program lies latent in the run-time objects. Patterns can capture the latent mental models in that structure. The object structure is obvious, but is too complex to call an *architecture*: it is like saying that the architecture of Chartres lies in its stones and bricks.

But, in fact, we do have the patterns! The first one in fact emerges from classical object orientation. The class structure is a good representation of the recurring commonalities among sets of objects at run time from the perspective of what the system *is*, if we set aside their aggregate behavior.



And, as we showed above, the role structure is a good representation of the rhythms of execution among sets of objects involved in a use case, from the perspective of what the system *does*. These two views capture the end user mental model. Having achieved that, we are much closer to the original goals of object-orientation than we find in the contemporary understanding of the paradigm.



DCI: Data, Context, and Interaction

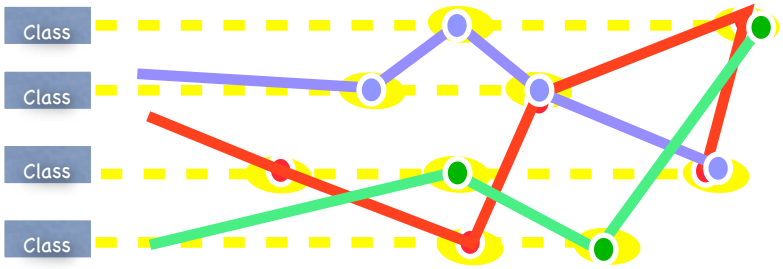
DCI (Data, Context, and Interation) is Trygve Reenskaug's new paradigm for software development, and it recalls many of the ideals that patterns and Agile are concerned with. (Trygve Reenskaug and Jim Coplien, *The DCI Architecture: A New Vision of Object-Oriented Programming*, http://www.artima.com/articles/dci_vision.html) How does DCI work? Let's focus first on the programmer mental model. Programmers have to carry out the two processes of designing what-the-system-is and what-the-system-does. For the former, we can use good old-fashioned domain analysis, or we can elicit the help of user experience people to mine end user mental models. The programmer can encode those concepts into classes. Those classes might even be organized into shallow classification hierarchies using inheritance, if the domain is sufficiently complicated. That's where the patterns of structure live. Most of the patterns in the published software literature fall into this category.

Second, the programmer can encode the analysis of what-the-system-does in new concepts called *roles*. Above, we found roles such as source account and destination account which were distinct in kind from concepts like Savings Account and Checking Account. While Savings Account and Checking Account are classes, source account and destination account are roles. These roles have methods that describe how the system executes, in time. That's where the patterns of events live. Few, if any, published software patterns capture this side of design. That's remarkable given the root of Alexander's pattern theory in the patterns of events.

We capture the snippets of this execution in roles, but we need to group roles together according to their patterns of enactment in a running system. As with all patterns, these patterns offer a solution only in some context. Alexander defines a pattern as a solution to a problem in a context. In DCI, we group roles together within an object representing a use case. DCI uses the term *Context* to represent this object. The name is a coincidence. And yet, of course, it isn't.

So much for the programmer's view. The programmer represents this view in code. However, this still isn't the architecture itself: it is a representation of the architecture that attains a high degree of compression of the information in the runtime system. We can compress it by encoding the commonalities across recurring instances of configurations of the forms both of the data and of the algorithms. That is, we find the patterns. At run time, there will be many class instantiations representing the concepts that the program deals with, sitting in memory, reflecting the concepts in the end user mind, awaiting the end user's bidding.

Once done *thinking*, the end user *does* something. The end user makes a specific request—a request for an enactment of a use case. The architecture captures use case scenarios in the Context and its roles. So the system chooses a Context suitable to the end user gesture or request. The Context knows how to find objects to play each role, and it "teaches" each object how to play its role in the use case by injecting the role's methods into the object. In some languages (Ruby, Python) this happens at run time. In others (C++, Scala) it happens at compile time. The resulting *objects* now have the knowledge to carry out the use case.



Note that this injection is done object-by-object, not class-by-class. The architecture is the *form* of the interactions. It is realized, use case by use case, in the *structure* of the association between roles and objects.

Back to the Future

First, it was Alexander's patterns, and second, the Agile discipline, that brought the human element of architecture to the forefront of modern software design discourse. Each of these ideals has played out in a flawed way in software: patterns, in their loss of systems thinking, and Agile, in its consideration of form. The DCI architecture is resurrects much of the original vision. It is more than just a software framework and is more than just a pattern language whose form captures mental models. It is a paradigm that captures a worldview that celebrates the human element of computer software.

Let's re-visit the original vision of the Hillside Group for patterns, and compare it to the provisions of DCI:

HILLSIDE VISION	DCI
• Patterns have brought to life that software is art, bifurcating the industry into pseudo-scientists and artisans	• The OO world is still about methods and GUI-builders. DCI and MVC focus on mental models and people
• They have brought a new programming language, and have enriched the vocabulary used by others.	• Qi4j is providing DCI functionality for Java and is paving the way for Java evolution. New languages are in the wings

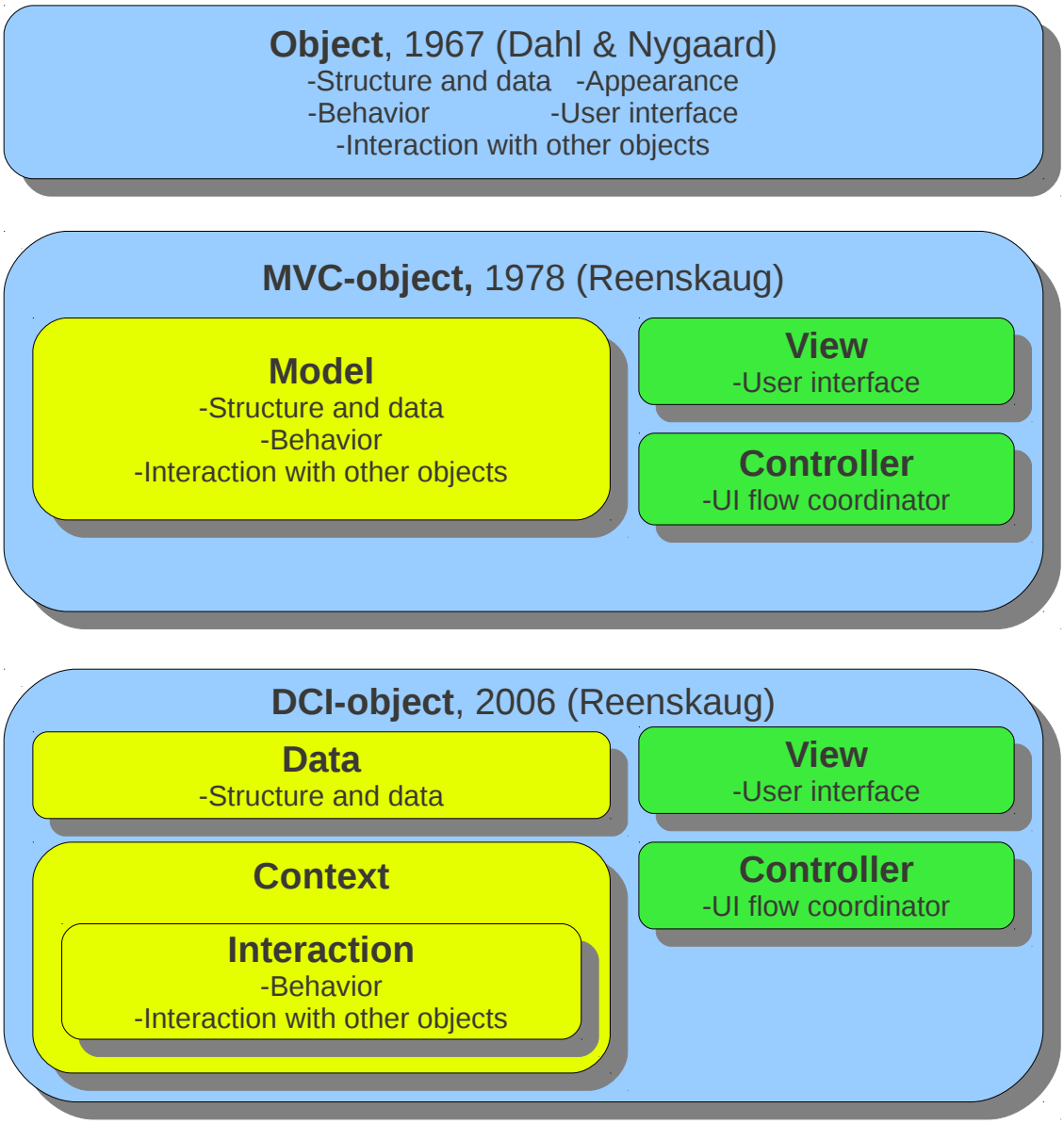
HILLSIDE VISION	DCI
• Patterns' major use is in neural networks, where they are used to describe high-level firing patterns and distribution of computation in emerging highly distributed systems.	• DCI is motivated by Kan's vision of a highly parallel machine of objects, moving us towards the vision of programming objects as extensions of the human mind, rather than classes
• Patterns have caused three major universities to shift software from their engineering and science programs, into liberal arts, with concomitant changes in curriculum (art history is now a prerequisite for a comp sci degree)	• MVC and DCI have returned human interfaces and mental models to their rightful place, with attentiveness to the aesthetics and comfort of human / computer interaction rather than just technical excellence

DCI also fulfills the Agile vision:

AGILE MANIFESTO	DCI
Individuals and interactions over processes and tools	Directly, DCI helps developers communicate with users in terms of their mental models of the world instead of computer-ese
Customer collaboration over contract negotiation	It provides a short path from user mental models to the code, which makes it possible to use tight feedback loops so the code converges on user expectations
Working software over comprehensive documentation	DCI makes it easier for us to reason about forms that are important to the user, such as the form of the sequencing of events toward a goal in a context, and to raise our confidence that the code does what we (and ultimately, the end user) want it to
Responding to change over following a plan	It catches change were it happens and encapsulates it, rather than spreading it across the architecture (as would be the case if we distributed parts of the algorithm across existing classes)

Symmetry Breaking Redux

DCI is an evolution of object orientation. As we look back on the past 40 years of history, we can look at the major advances in object-oriented design as a process of differentiation of form. Risto Välimäki offers an illustration of object orientation's path from basic objects through the realization of DCI. The original worldview comprised objects that encapsulated methods and data. These systems had *some* texture in their form: local symmetries across data and methods, as well as the local symmetries of inheritance hierarchies. Both of these forms represent symmetry; symmetry represents a pairing of invariance and potential for change across sets of things. So a shape would contain data to represent its area; the method area represents the same concept, though each one transforms the concept (or part of it) into a representation of bits suitable either for storage or computation. Inheritance hierarchies exhibit a higher-order symmetry: two classes in a hierarchy can each instantiate objects such that can interchangeably be bound to the same identifier elsewhere in the program. This symmetry is called the *Liskov substitutability principle*, or sometimes, just *polymorphism*.


Object, 1967 (Dahl & Nygaard)
-Structure and data -Appearance
-Behavior -User interface
-Interaction with other objects
MVC-object, 1978 (Reenskaug)
Model
-Structure and data
-Behavior
-Interaction with other objects
View
-User interface
Controller
-UI flow coordinator
DCI-object, 2006 (Reenskaug)
Data
-Structure and data
Context
Interaction
-Behavior
-Interaction with other objects
View
-User interface
Controller
-UI flow coordinator

However, this symmetry existed only in the code, and only for the programmer. MVC-U added local symmetries that had relevance at run time and which therefore related to the end-user view rather than the programmer view. It redistributed the object's symmetry into a visual component for the display and a computational component for the computer, the latter just being an object of the previous generation, now called the *Model*, with some logic to maintain invariance with its visual counterpart (the *View* and the *Controller*). This pushed the symmetry outside the machine to squirt through into the user world.

One way to view DCI is that it further redistributes the local symmetries of the Model into parts that are differentiated for system-level action and for local concerns. We introduce the *Context*, which is in fact a composition of the symmetries of role methods that formerly existed as class methods. In accordance with Alexander's model of the evolution of form, this redistribution of symmetry doesn't correspond to a decomposition tree: instead, the Context cuts across several objects. Alexander maintained that such entangled structures must be at the foundation of any system that is "alive." (Christopher Alexander, *A City is Not a Tree,* Architectural Forum, 122(1), pp. 58–62, April 1965) What Risto's diagram does not show is the breaking of symmetry across the static code structure and the run-time code structure.

In this sense, we make the (albeit speculative) argument that DCI could represent a natural evolution of object structure into its next generation. We are reminded that the simplistic, pure symmetry of early object orientation is unlikely to be right:

> Living things, though often symmetrical, rarely have perfect symmetry. Indeed perfect symmetry is often a mark of death in things, rather than life. I believe the lack of clarity in the subject has arisen because of a failure to distinguish overall symmetry from local symmetries.

We can also reiterate Alexander's words from Chapter 9 of *Timeless Way of Building*, applied now to computer software instead of buildings:

> It is a geometrical image, it is far more than the knowledge of the problem; it is the knowledge of the problem, coupled with the knowledge of the kinds of geometrics which will solve the problem, and coupled with the feeling which is created by that kind of geometry solving that problem.

And if we compare Risto's diagram with any equivalent UML representation, we can likely claim Alexanders stipulation about *ma*:

> And finally, of course, I want to paint a picture which allows me to understand the patterns of events which keep on happening in the thing whose structure I seek. In other words, I hope to find a picture, or a structure, which will, in some rather obvious and simple sense, account for the outward properties, for the pattern of events of the thing which I am studying.

Perhaps, for the first time in computing, the deep principles of object orientation have come to fruition.

Conclusion

The original goals of the pattern community were noble, and derived from a broad vision of human dignity and service of the architect Christopher Alexander. The pattern community continues to thrive as a culture of mutual support and of value in sharing good ideas. However, patterns themselves have become vulgar. Even the best pattern never mastered the Japanese concept of *ma*: of integrating form and function, space and time. However, the pattern values live on in the Agile community. DCI is one of the most concrete manifestations of the architectural expression of these principles today — including the form of function.