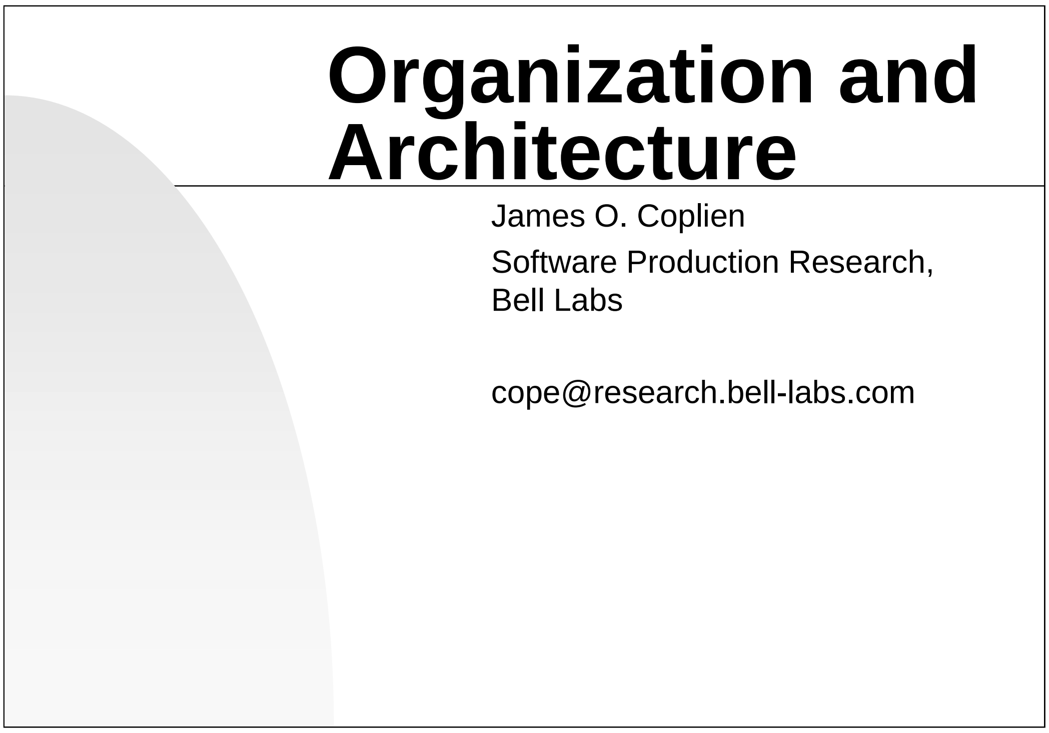


Organization and Architecture

James O. Coplien

Software Production Research, Bell Labs

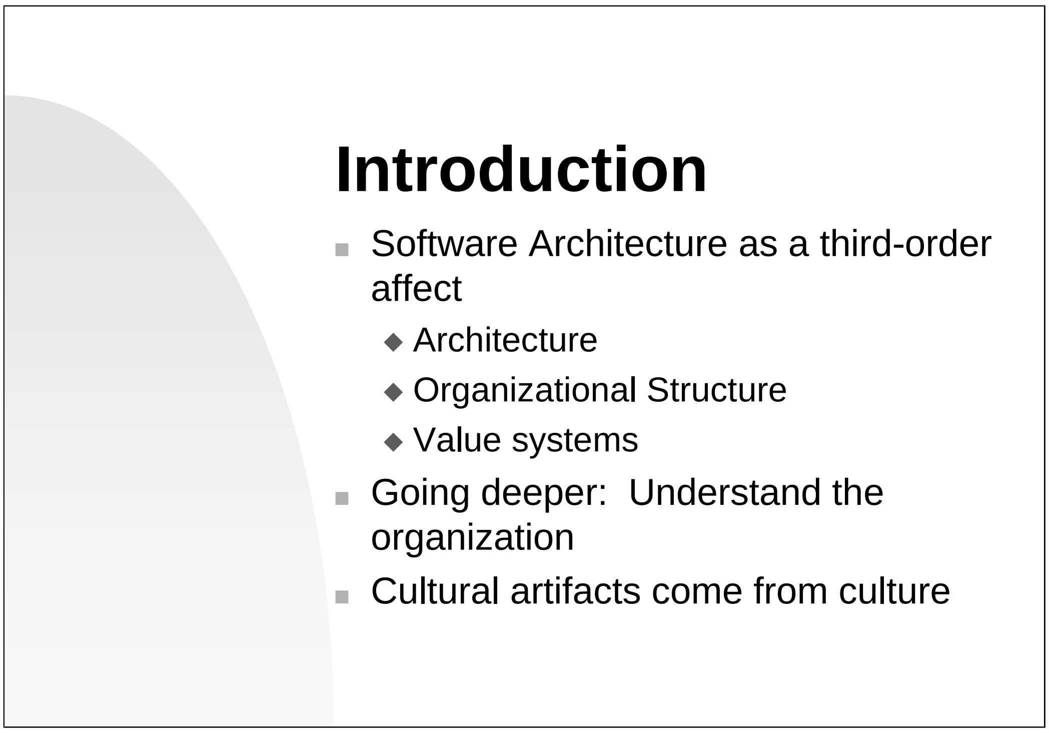
cope@research.bell-labs.com

In 1968, Mel Conway postulated that the structure of any information system is isomorphic to the structure of the organization that built it. Since then, we've been through several changes in technology and design paradigm on the architecture side of software, and through many management fads on the organizational side of software. Has anything changed?

This talk looks at some of the important relationships between software architecture and software development organization. The talk is based in part on a career of anecdotal evidence, but it draws most directly on seven years of empirical research in organization structure. The talk addresses questions that loom important to software development today.

Is geographically distributed development feasible? Does code ownership help or hurt? What is the architect's role, and how should expertise be allocated to roles in development organizations? If Conway's law holds absolute sway, what does this portend for new design paradigms?

CHOOSE Forum, 16:30 - 18:00, 11 March 1999, University of Bern, Main Building, Room 31.



Introduction

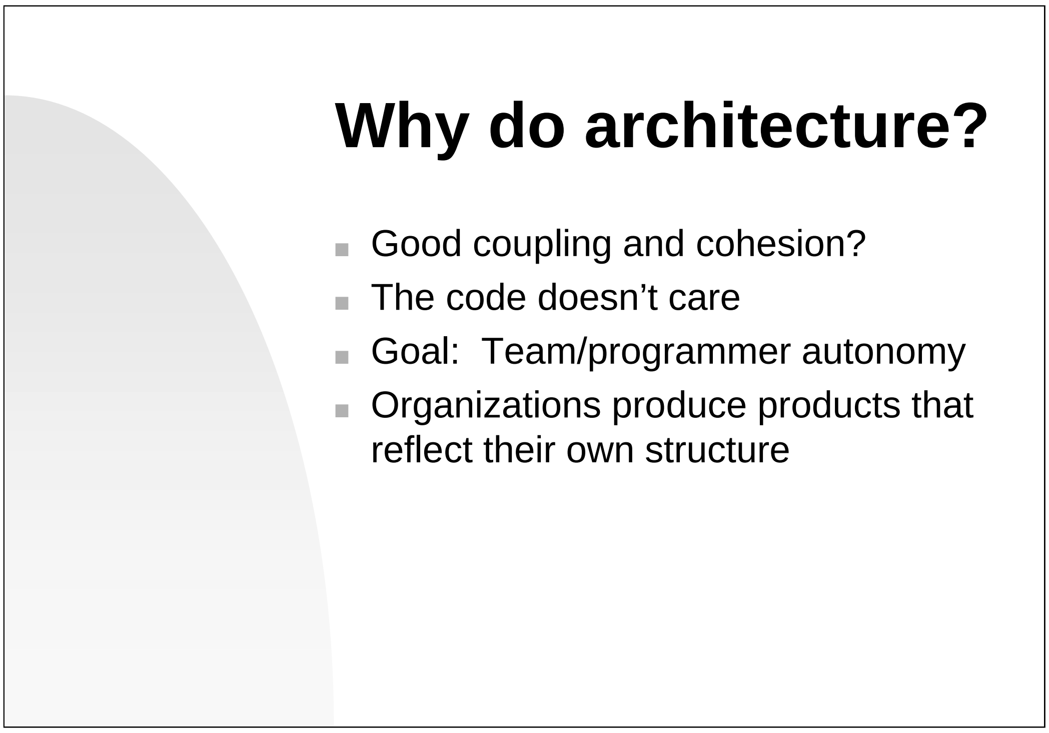
- Software Architecture as a third-order affect
  - Architecture
  - Organizational Structure
  - Value systems
- Going deeper: Understand the organization
- Cultural artifacts come from culture

Computer science loves to focus on design and things related to design: languages that express design, paradigms that shape design, methods that guide design. The products of design are an architecture and an implementation. Because of our preoccupation with these deliverables, we focus on the tools and techniques that support their creation and nurturing.

Yet architecture is less an echo of the tools and methods that create it than of the organization that built it. This parallelism is called Conway's law, about which we'll say more in following slides. More generally, the product of any group reflects the structure of the group. The Tower of Babel couldn't be built because it had to be a single structure, yet its builders were structured into several groups that couldn't communicate with each other.

Even more important — but beyond the scope of this talk — is the underlying value system. Value systems generate structure, and structure generates process. This layering is well known to cultural anthropologists and sociologists, students of human behavior.

Software is an intensely social and human activity. It bears study through the eyes of the human sciences much more than from the perspective of so-called computer science. In this talk, we'll examine some empirical findings that underscore this perspective.



Why do architecture?

- Good coupling and cohesion?
- The code doesn’t care
- Goal: Team/programmer autonomy
- Organizations produce products that reflect their own structure

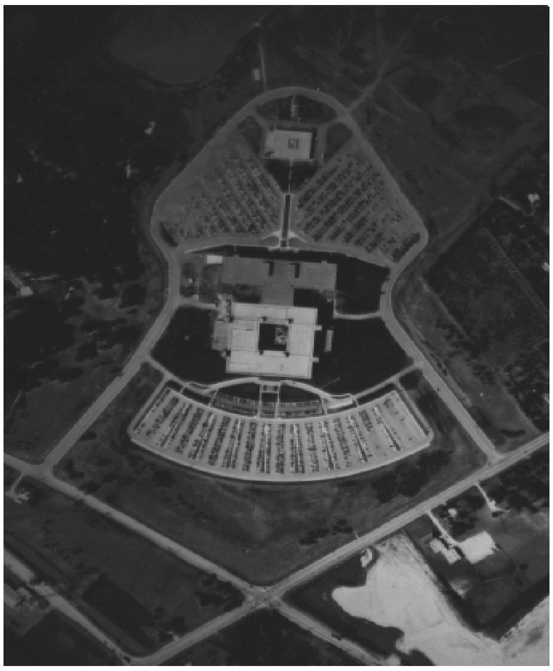
Why do we care about architecture, and what do we focus on? We try to minimize coupling and maximize cohesion. The original reasons for this focus are almost lost to history: the goal was to allow groups to work independently on their modules. The code doesn’t care whether it’s cohesive or not. In fact, highly coupled code tends to perform better than code with more layers of abstraction.

We need to focus beyond cohesion and coupling in their own right and look to the needs of the organization. Organizations produce products that reflect their own structure.

PUBLIC
BELL
SYSTEM
TELEPHONE

At one time, this was one of the most recognizable trademarks in the world. In the beginning, the Bell System had beauty, elegance, symmetry, monopoly...

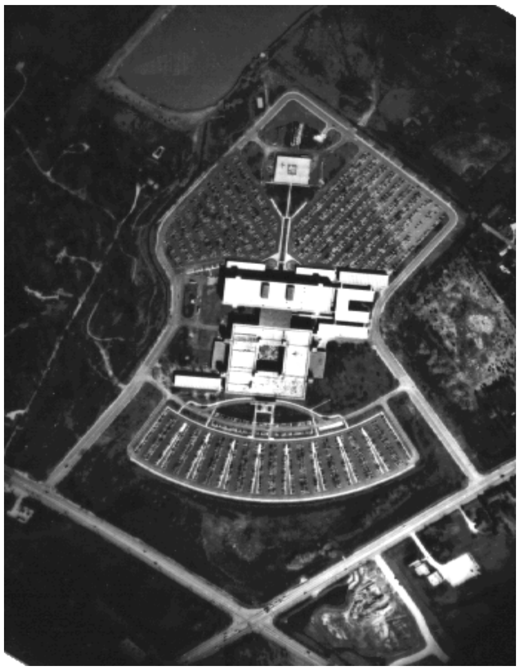
This is an aerial photograph of a plot of ground near Naperville, Illinois, in 1963.



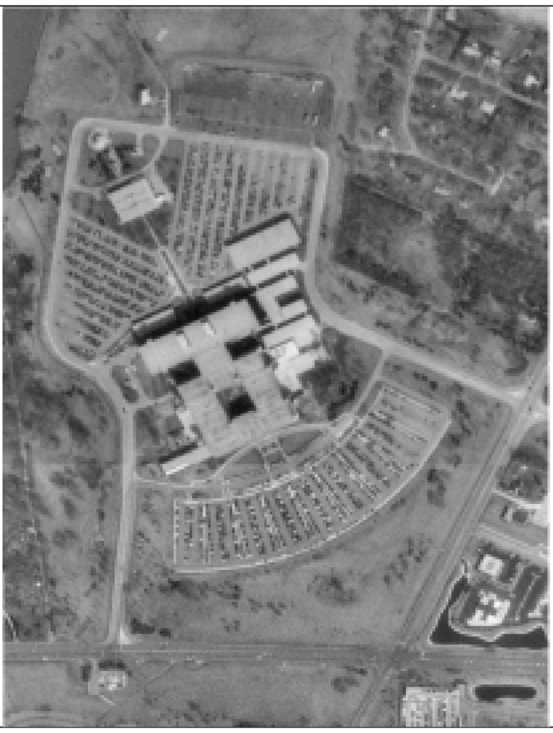
By 1976, a large company had built a large R&D site on the property.

By the way, I've shown these slides in an auditorium in the above building, and only about 20% of the people in the audience knew about this architectural artifact of the building.

If you work inside a large system whose major architectural structures are not known to you, bad things happen...

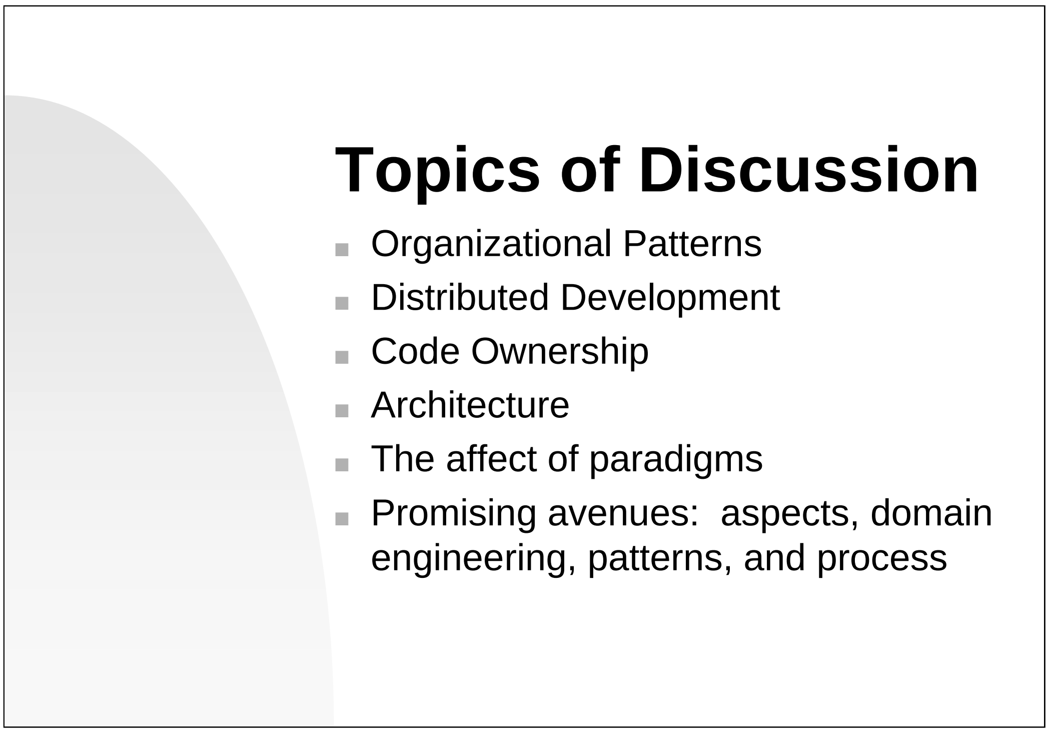


And, by 1988, the architecture had lost much of its symmetry.



By 31 March, 1994, it was even more misshapen — and one can see the advent of a new parking lot in the back.

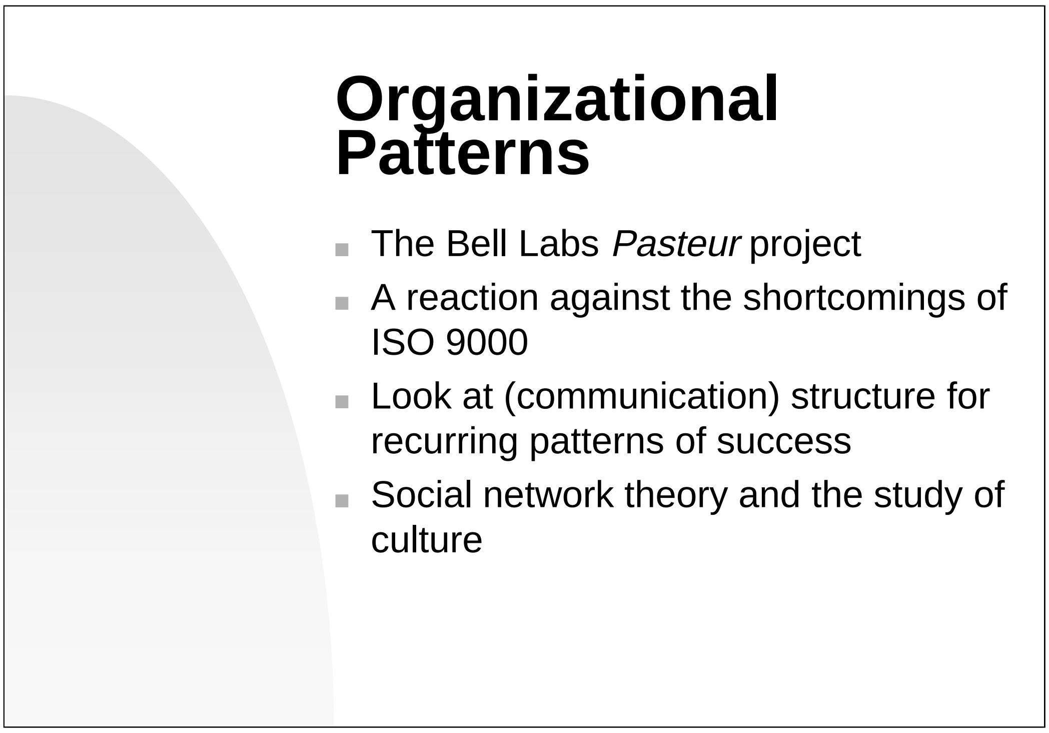
Today, there is yet another parking lot in back behind the one being built here, and there are two new buildings below the bell clanger.



Topics of Discussion

- Organizational Patterns
- Distributed Development
- Code Ownership
- Architecture
- The affect of paradigms
- Promising avenues: aspects, domain engineering, patterns, and process

In this talk, I'll cover what I've found to be the important links between organization and architecture. The talk is based in organizational pattern research at Bell Labs over the past 8 years. The major topics of the talk are distributed development, code ownership, architecture, the affect of paradigms, and promising new technologies that embrace the challenges of structuring organizations in large projects.

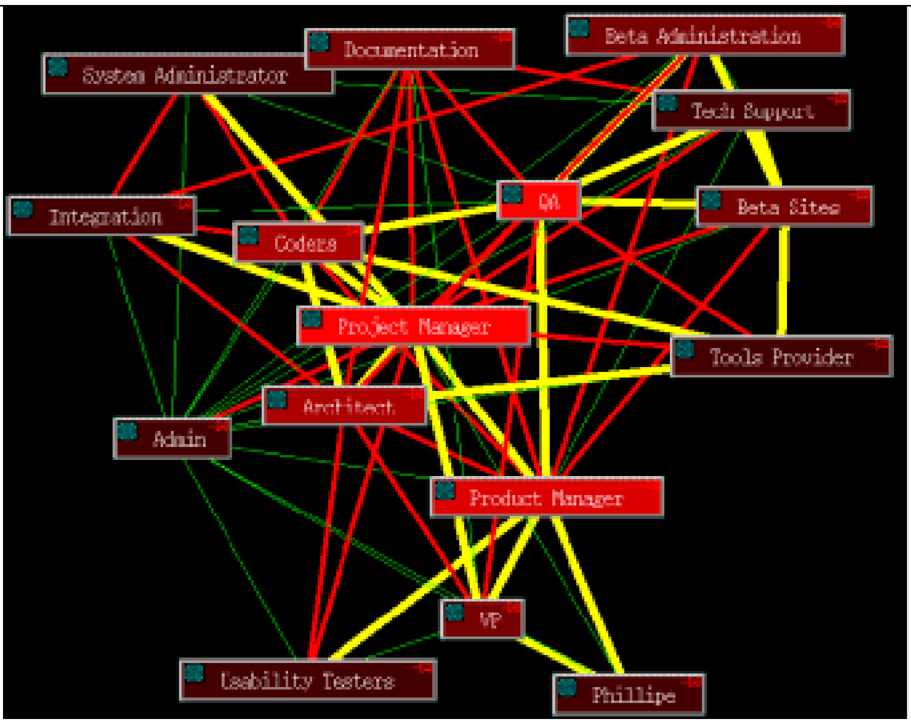


Organizational Patterns

- The Bell Labs *Pasteur* project
- A reaction against the shortcomings of ISO 9000
- Look at (communication) structure for recurring patterns of success
- Social network theory and the study of culture

The Pasteur research project at Bell Labs was a large, long-running effort to document the communication structure of software development organizations and associated professional organizations. The project gathered empirical data from dozens of development projects, using a role-playing technique based on CRC cards. This technique captured the instrumental structure of the organization — not what appeared on the organizational chart.

We found that the patterns in these pictures spoke more about the health of the organization than any ISO 9000 process spec did, and we started cataloging and exploring the patterns. We soon discovered that we had, in essence, reinvented social network theory, and we adopted many of that discipline's metrics and measures.

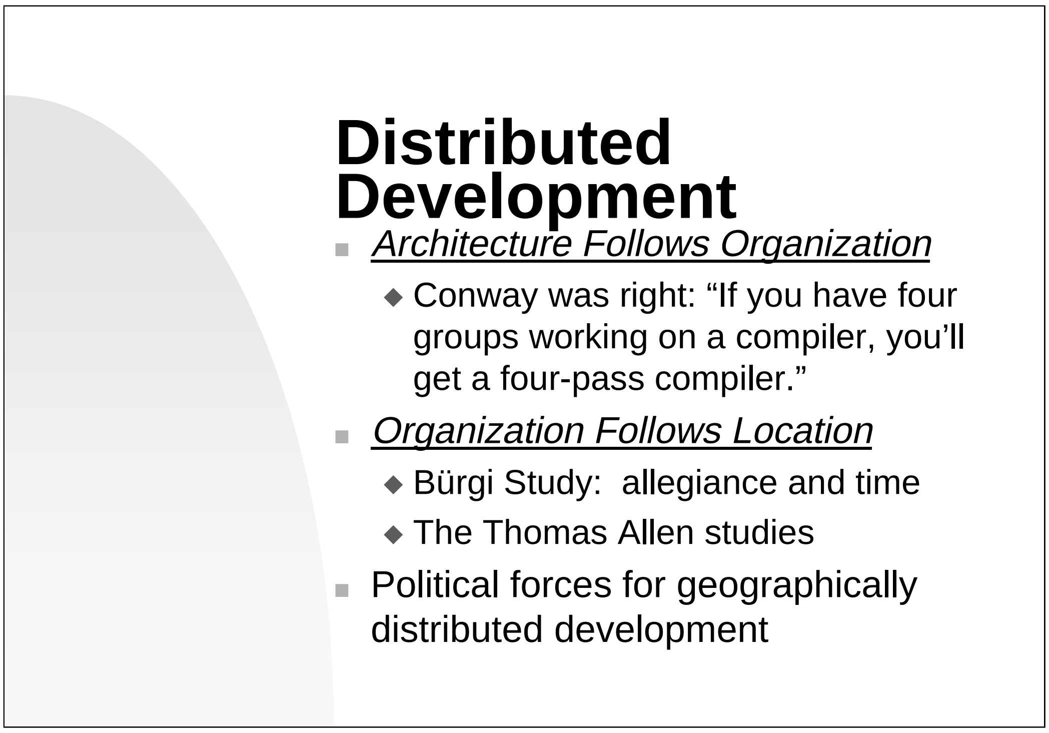


This is the social network of the Borland QPW project. Each box corresponds to a role in the process. The brighter red a box is, the more central it is to the communication network. The diagram is called a *social network diagram* or *adjacency diagram*.

Yellow lines indicate paths of intense communication; red lines, paths of moderate communication; green lines, paths of light communication. These lines come from an aggregate weighting of individual interactions between roles as related by the participants. The measure is subjective, based on frequency, importance, and bandwidth.

The roles are placed on the page using a force-based two-dimension relaxation algorithm. The most central roles (those most engaged in communication) move to the center; those that are most decoupled move to the outer edges.

This graph is more devoid of strong patterns than other pictures we have. In fact, the evenness is the sign of a successful organization. We can see a central architect and a highly engaged QA function, the presence of a patron, and other configurations indicative of patterns we find in successful organizations.



Distributed Development

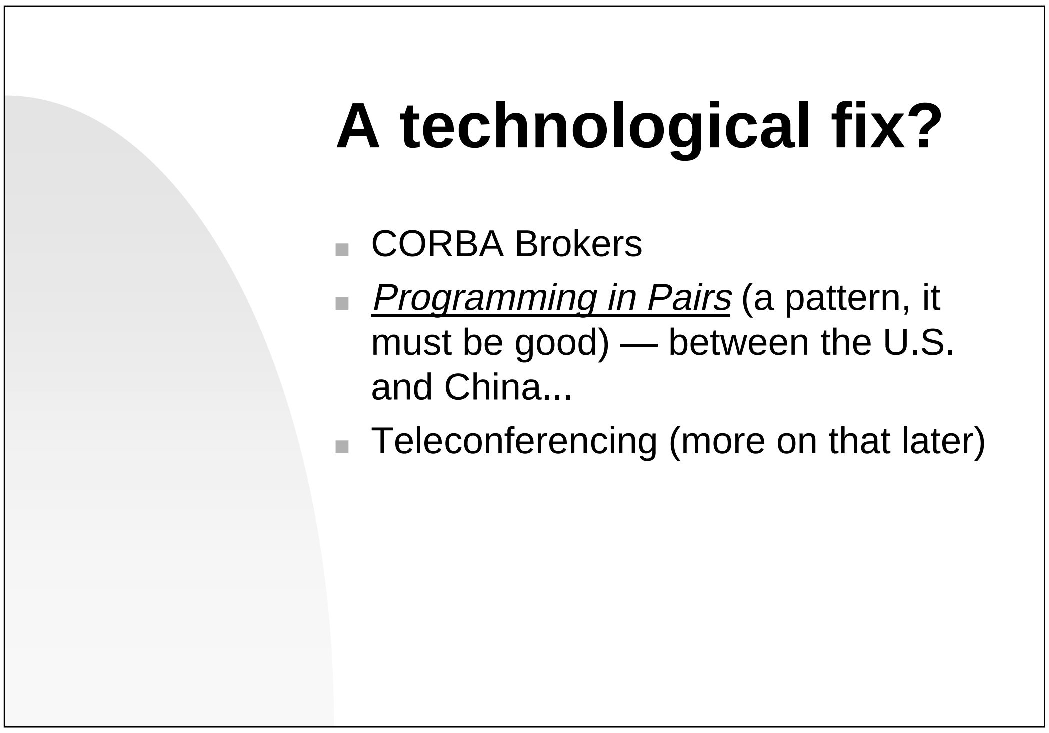
- *Architecture Follows Organization*
  - Conway was right: “If you have four groups working on a compiler, you’ll get a four-pass compiler.”
- *Organization Follows Location*
  - Bürgi Study: allegiance and time
  - The Thomas Allen studies
- Political forces for geographically distributed development

Large projects divide into multiple groups. The resulting product will always reflect the structure of those groups — an observation made many years ago (1968 or earlier) by Mel Conway. Our organizational studies found that *Architecture Follows Organization* (pattern names are underscored and italicized in this talk): just a restatement of Conway’s Law. We also found that *Organization Follows Location*, no matter what the organizational chart says.

In a study of geographically distributed development (across the Atlantic Ocean), anthropologist Peter Bürgi found that the dominant forces driving interaction between cultures were allegiance (local) and time (shear).

Thomas Allen at Sloan School has mapped social distance as a function of physical distance; it’s very nonlinear. A staircase is an almost impenetrable barrier. Even a hallway corner is bad. In short, distribution destroys communication.

This is a crucial consideration in today’s markets, where vendors are encouraged to make a vested economic presence in countries where they wish to establish a market.



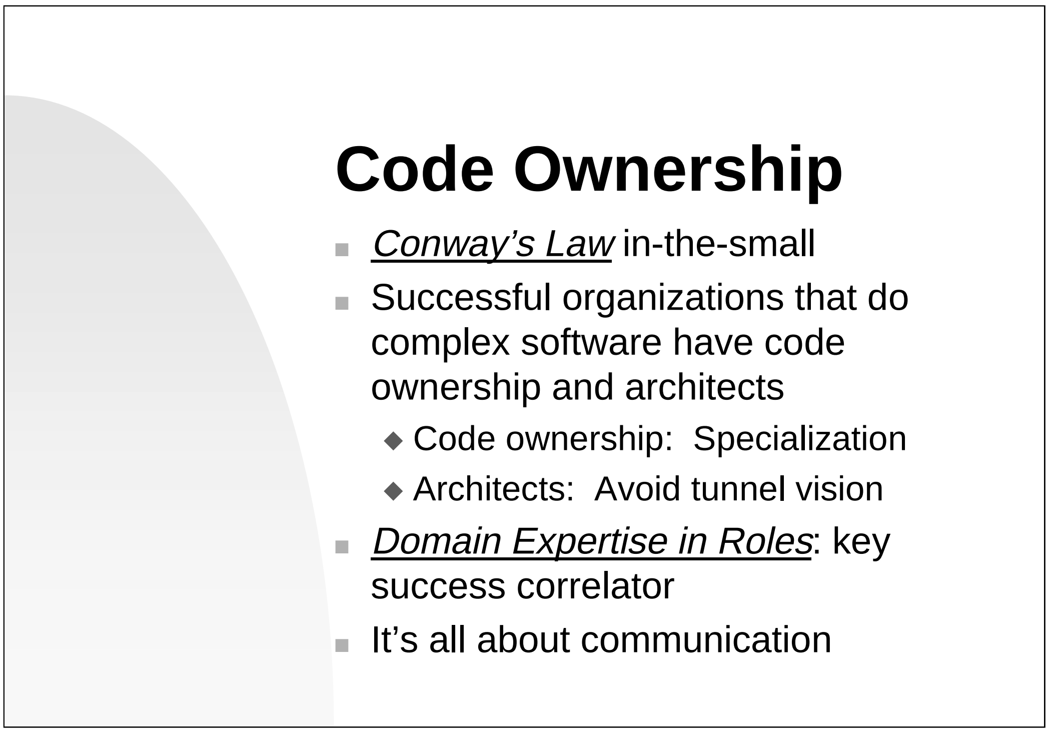
A technological fix?

- CORBA Brokers
- *Programming in Pairs* (a pattern, it must be good) — between the U.S. and China...
- Teleconferencing (more on that later)

Of course, being engineers, we feel that technology triumphs over all. I watched one project try to overcome geographic distribution (three states and two countries) using a CORBA broker; the project because so dependent on the idiosynchracies of the broker that it died under its own weight.

Another favorite was a project where management decided that *Programming in Pairs* was a good idea, so they paired good programmers in one country with others on the other side of the Pacific Ocean…

Teleconferencing is another supposed technological solution to an organizational problem — but more about that later.

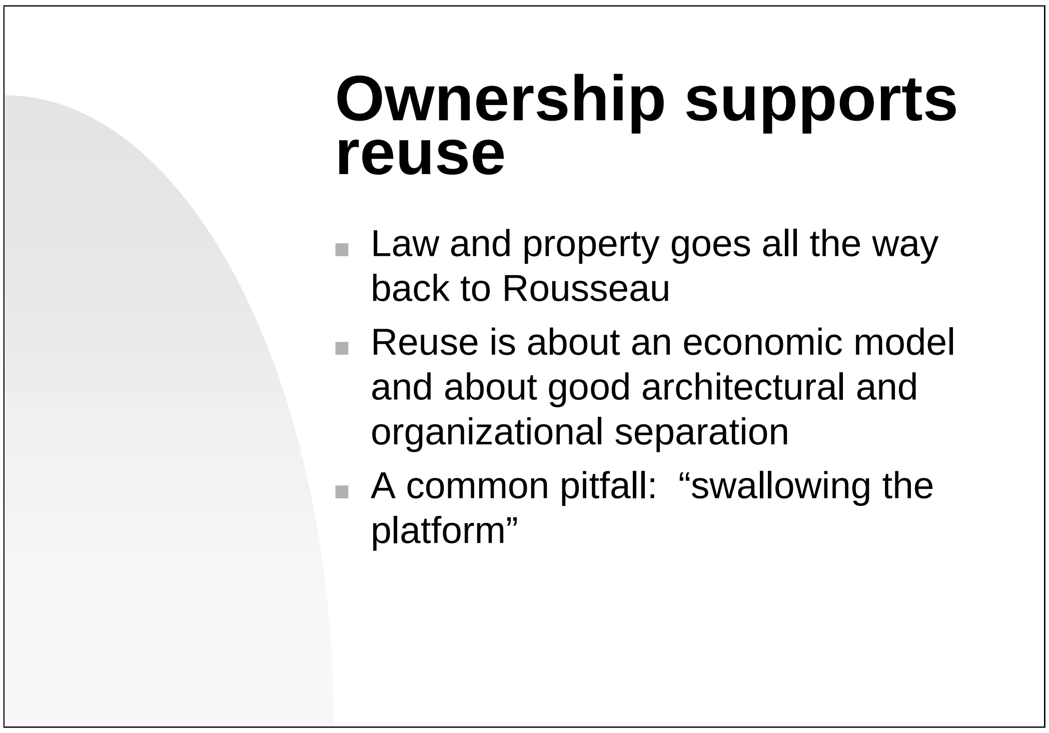


Code Ownership

- *Conway's Law* in-the-small
- Successful organizations that do complex software have code ownership and architects
  - Code ownership: Specialization
  - Architects: Avoid tunnel vision
- *Domain Expertise in Roles*: key success correlator
- It's all about communication

Large projects that thrive over time usually practice code ownership. It's a form of Conway's Law in the small. Code ownership builds on specialization: when the domain expertise is too broad for the mind of a single person, code ownership makes it possible to leverage the separate skills of multiple individuals. Architects provide the high-level glue that avoids tunnel vision.

The key expertise isn't in the solution domain technology, but in the application or business domain.



Ownership supports reuse

- Law and property goes all the way back to Rousseau
- Reuse is about an economic model and about good architectural and organizational separation
- A common pitfall: “swallowing the platform”

Reuse is one of the Holy Grails of modern software, and it is largely a myth. Having little to do with technology, reuse is about organizational, political, economic, and social issues. Yes, there is an architectural component, but it’s subservient to the economics.

One pattern I’m seeing more and more is that:

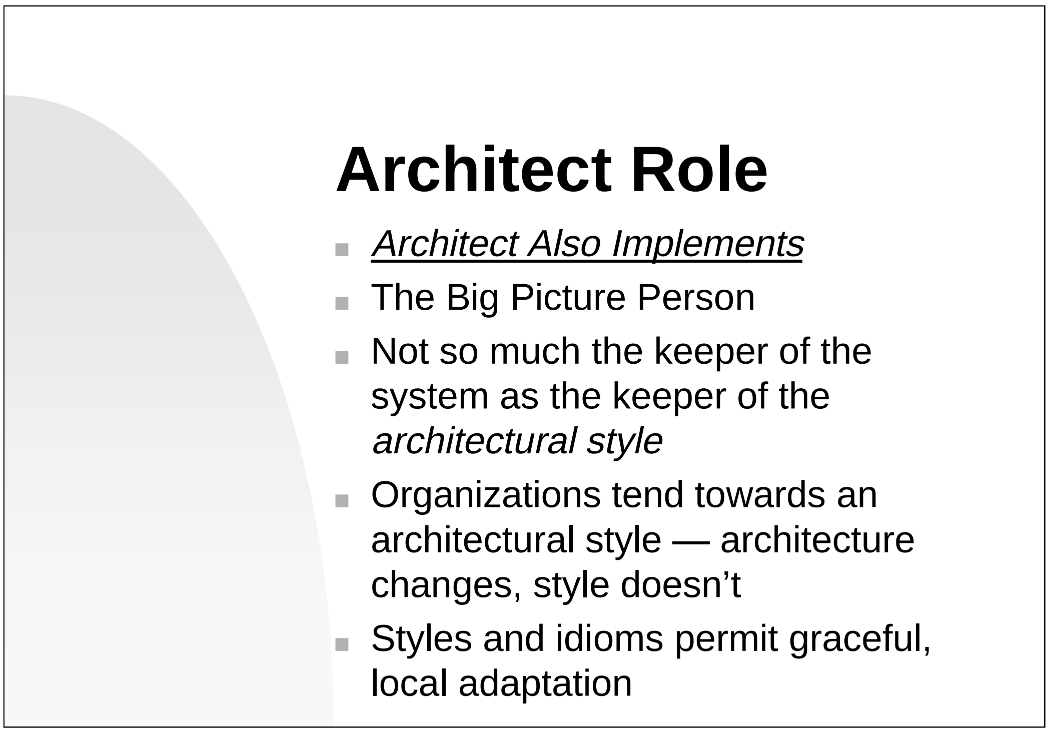
1. An organization sets out to build reusable assets;

2. They obtain a customer;

3. The customer insists on control and swallows the organization building the reusable artifact;

4. The artifact becomes so contextualized in the parent organization that it can’t be reused anywhere.

Only by keeping the organizations separate can reuse be achieved. It’s not a matter just of architecture and design.



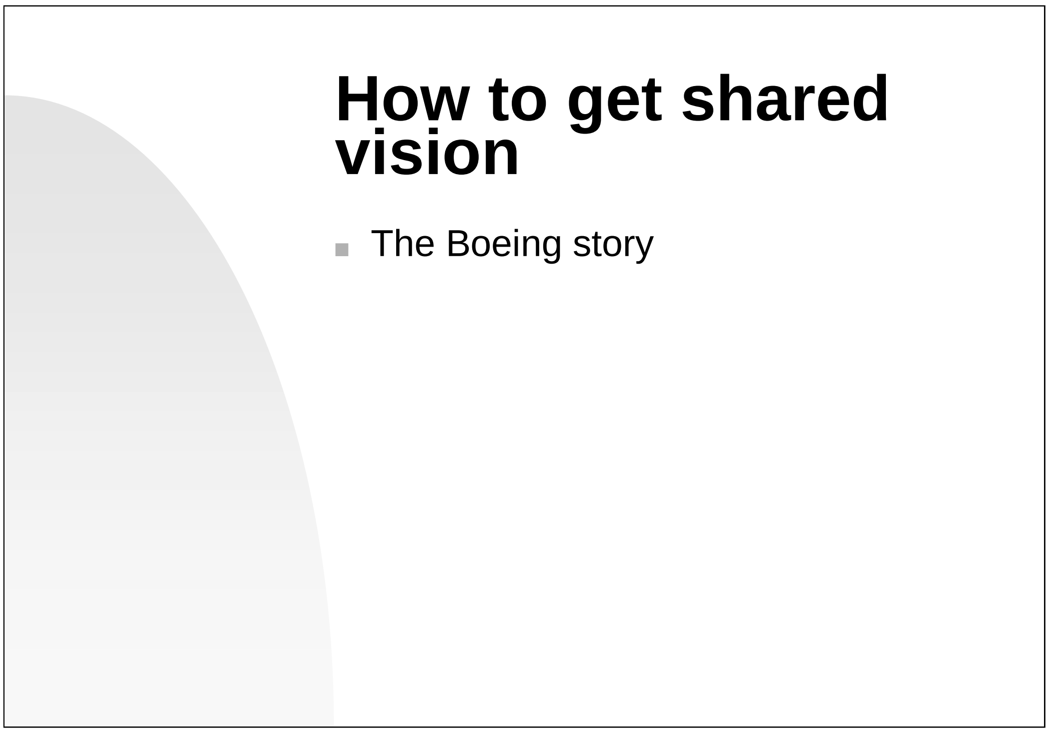
Architect Role

- *Architect Also Implements*
- The Big Picture Person
- Not so much the keeper of the system as the keeper of the *architectural style*
- Organizations tend towards an architectural style — architecture changes, style doesn't
- Styles and idioms permit graceful, local adaptation

As mentioned before, architects make up in the broad view what specialists lack in their local view. But the architect is not a deity nor a repository of all knowledge about structure. The architect is just the master builder, the keeper of the pattern language for the system structure: the keeper of the architectural style.

A development organization that follows an architectural style. A large project in Lucent started in 1978 and first deployed in 1982 with about 26 subsystems comprising a total of a few hundred thousand lines of code. The system had an architecture that followed a certain architectural style. Today, that system still thrives — at 60 million lines of code, still growing and serving larger markets. Can one imagine that any vestige of that original architecture remains anywhere hidden in those 60 million lines? Probably not — but much of the style is the same.

Styles, and the idioms/patterns that go with them, permit designers to work autonomously in a way that leads to graceful evolution. There may be replication, but that, too, gets cleaned up along the way in a healthy project — which is why *Code Ownership* can't be too strong.

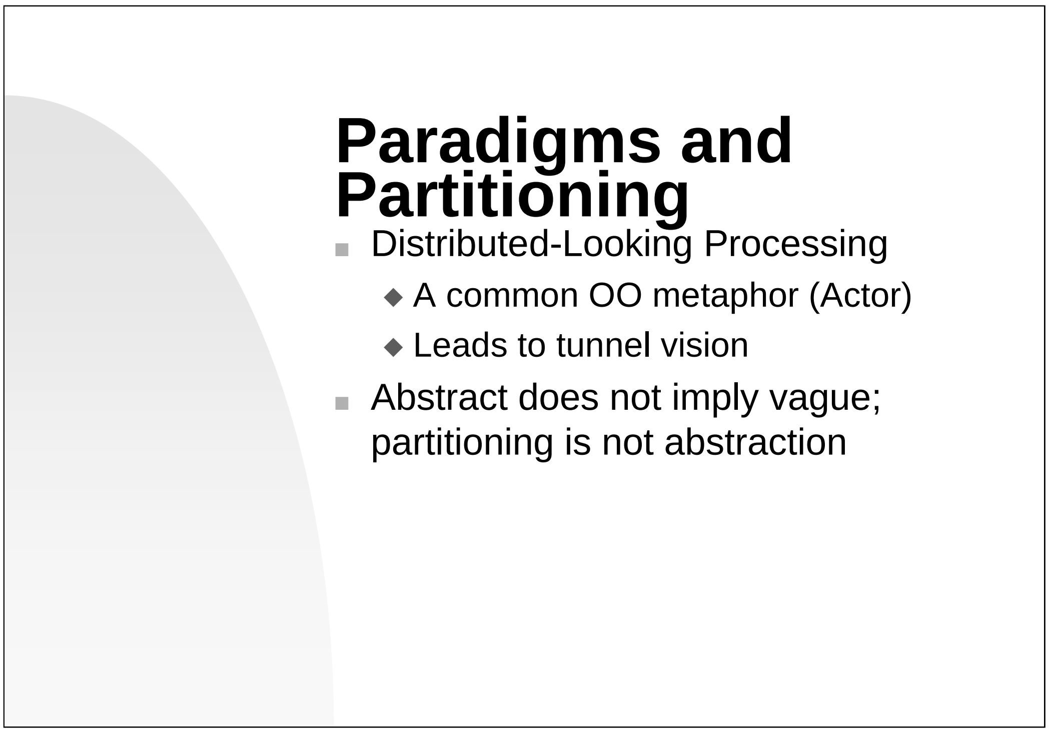


How to get shared vision

- The Boeing story

Larry Constantine tells the story of the Boeing 777 project, which felt it was important to gather all project members under one roof at one time, several times during the project. I think the numbers were close to 10,000 people. Well, if you're an airline company, you find a place big enough to do that: a large airline hangar. And you have the means to bring them there: fly them from all around the world. This coming together led to an intangible sense of common purpose, and led to relationships that eased communication throughout the project.

If one were a telecommunications company, one wonders if it comes down to a 10,000 person conference call...

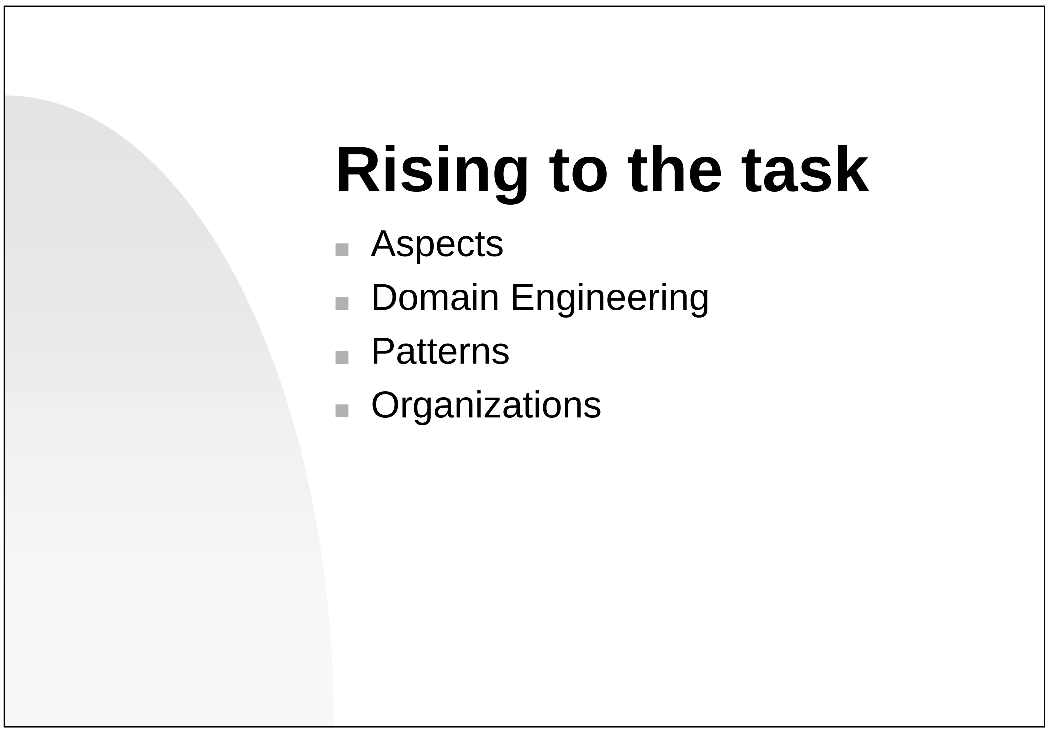


Paradigms and Partitioning

- Distributed-Looking Processing
  - A common OO metaphor (Actor)
  - Leads to tunnel vision
- Abstract does not imply vague; partitioning is not abstraction

The industry seems to hold onto a fascination with distributed processing. One often finds analysis model that assume infinite processors, ostensibly for the sake of not over-constraining the solution. However, this leads to a strange cargo cult style of programming that seems to have few practical benefits.

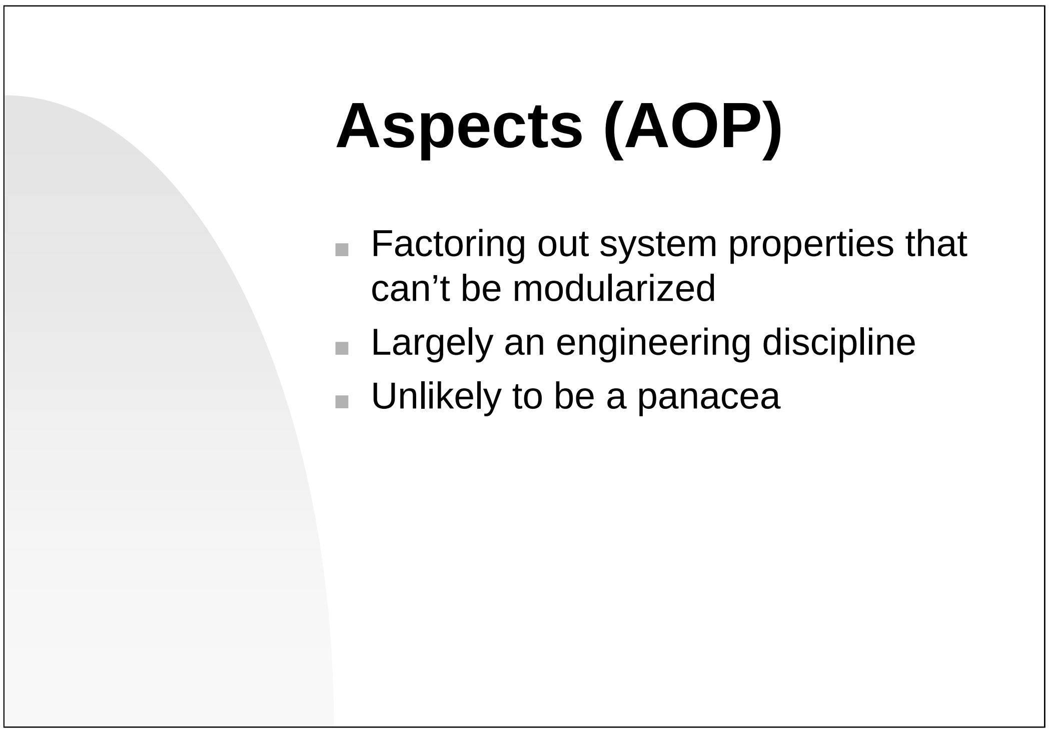
Early OO methods — and many facets of contemporary OO practice — emphasize this pseudo-distributed nature of the problem. Forcing distribution adds accidental complexity that isn’t germane to the problem.



Rising to the task

- Aspects
- Domain Engineering
- Patterns
- Organizations

Most of the recent advances in software design directly tackle these organizational issues by considering the parallels between organizations and the factorings supported by the technology at hand. Aspect-oriented programming separates out cross-cutting concerns that normally exacerbate group independence. Domain engineering overcomes the artificial coupling that arises from the application of one shape of partitioning criteria to another shape of problem. Patterns explicitly focus on wholeness, on systemic relationships. And first-class organizational practices can tackle many of these problems first-hand.

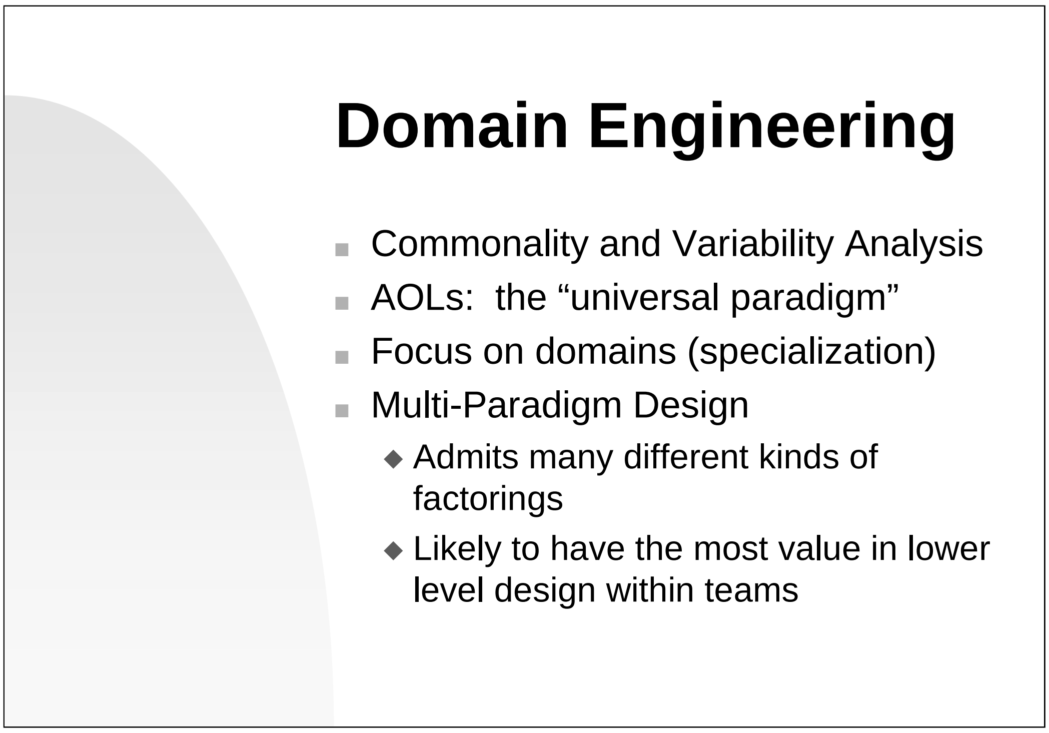


Aspects (AOP)

- Factoring out system properties that can’t be modularized
- Largely an engineering discipline
- Unlikely to be a panacea

Aspect-oriented programming provides “a better handle on managing cross-cutting concerns.” The idea is tantalizing: take any perspective of a system and factor it out as a self-contained entity. However, aspects can be only a stopgap measure, particularly for systems with many non-orthogonal “tops”. No matter how you cut the aspects, some of them will interfere with each other at the specification level.

AOP is likely to go a long way in extending the facilities of MOPs and of application generators to provide flexibility in complex designs, and for dealing with non-functional requirements. The organizational benefits can be substantial.



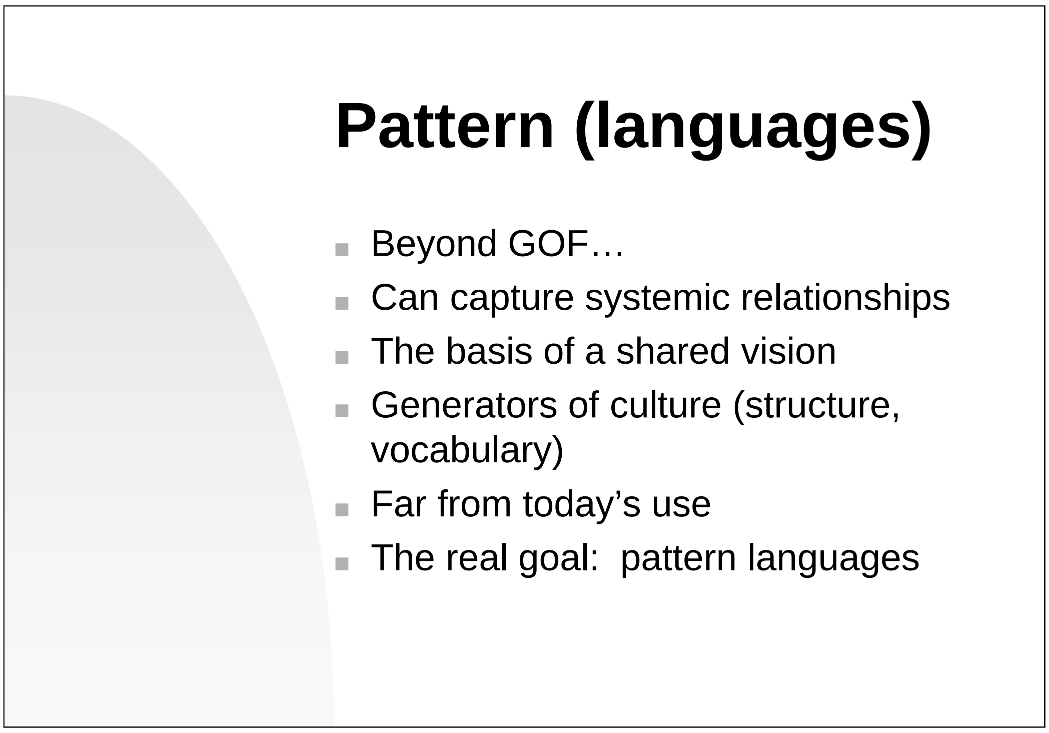
Domain Engineering

- Commonality and Variability Analysis
- AOLs: the “universal paradigm”
- Focus on domains (specialization)
- Multi-Paradigm Design
  - Admits many different kinds of factorings
  - Likely to have the most value in lower level design within teams

Domain engineering goes back to basics: how do we create abstractions, and how do we express them? Going to the primitive level of commonality and variability analysis, we go beyond paradigm into a meta-paradigm world where, in theory, everything can be custom-fit. Indeed, one possible output of domain engineering is an application-oriented language. But even without an AOL, domain engineering helps the designer divide things cleanly along lines of expertise — along *domain* boundaries.

Many projects try to impose an object structure on a problem no matter how good or bad the fit. The misfit between object partitioning criteria, and the natural creases in the problem, leads to dependencies between modules that wouldn’t be there if another paradigm were used. Multi-paradigm design optimizes the choice of paradigms to maximize autonomy of modules.

While multi-paradigm design most often is associated with a small number of primitive paradigms (object, procedural, functions and rules, or objects, procedural, overloading, parametric, etc.) it easily can be parameterized to deal with any solution domain technologies. Because the scope of most multi-paradigm methods is close to the programming language level, many of the benefits are likely to be at low levels.



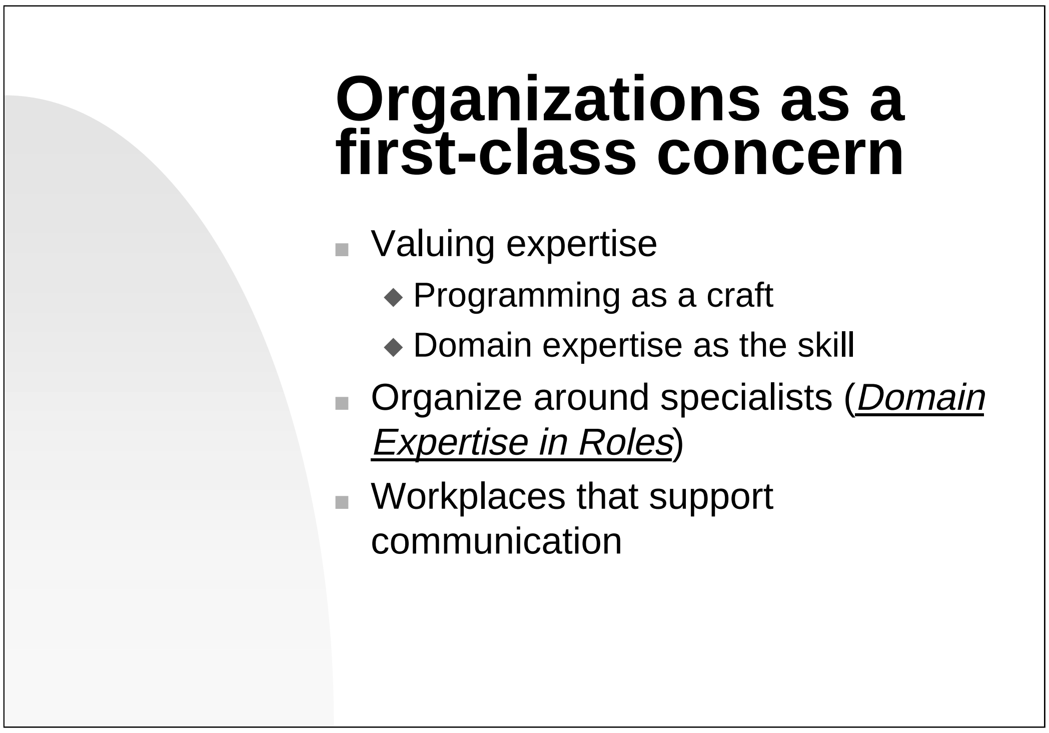
Pattern (languages)

- Beyond GOF…
- Can capture systemic relationships
- The basis of a shared vision
- Generators of culture (structure, vocabulary)
- Far from today’s use
- The real goal: pattern languages

By “patterns,” I don’t mean the things one finds in the Gang of Four book. Alexander’s vision for patterns is that they capture system-level relationships of whole structure that support quality of life. Patterns provide a shared vision of architecture that is impossible to communicate effectively in technological terms. Used insightedly, patterns are operative at a cultural level.

Several mature development culture have been using patterns (consciously or not) this way for years. In fact, anthropological literature has used the term “pattern” in this sense for many years, in the sense of a recurring cultural structure. Sadly, few of the new age pattern practitioners rise to this level: for most, patterns are just a cute form to document special exceptions or tricks.

Properly used, no pattern stands alone. The real value comes not from patterns, but from pattern languages. Patterns alone are a small increment beyond objects.

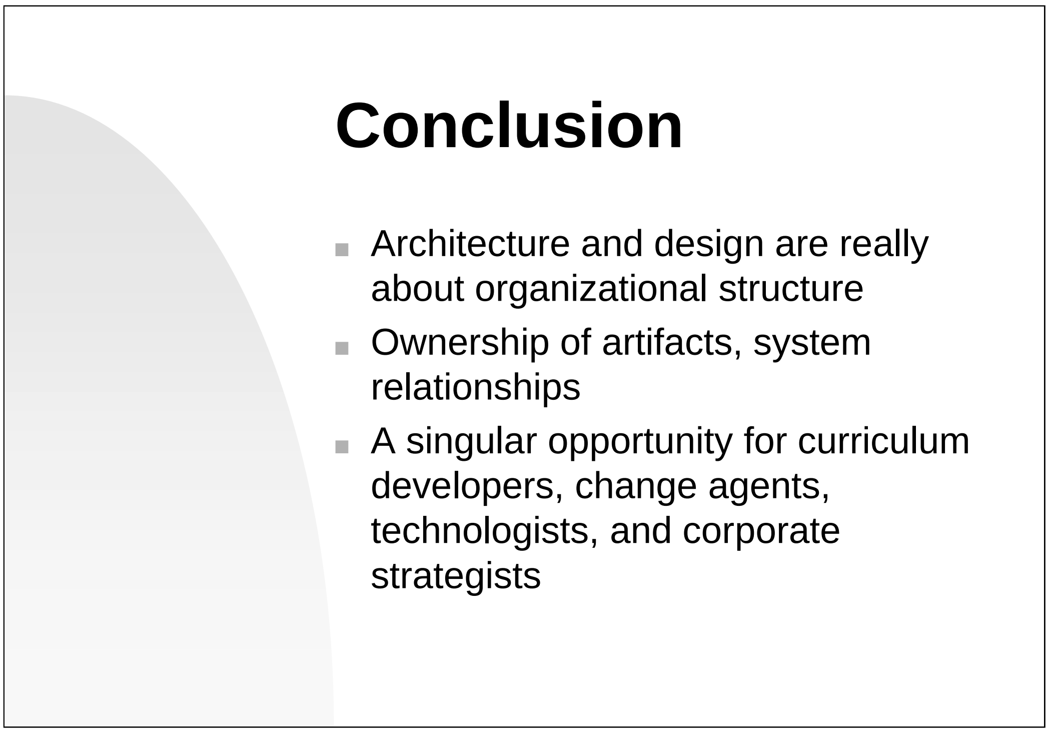


Organizations as a first-class concern

- Valuing expertise
  - Programming as a craft
  - Domain expertise as the skill
- Organize around specialists (*Domain Expertise in Roles*)
- Workplaces that support communication

Instead of attacking these problems architecturally, we can directly attack some of them organizationally. Organizing around expertise, around domains, instead of around other units, helps align architecture and organization. The expertise in question here isn't computer science or software engineering expertise, but business expertise.

It's also important to have workplaces that support communication. Groups that work together should be colocated. Careful attention should be paid to putting people on the same floor, in sight of each other. These seem like details, but they can make or break a project. For example, the existence of one long hallway (that went outside the building proper) between the marketing and engineering arms of a large west-coast company exacerbated the friction between those two divisions, and almost killed the company.

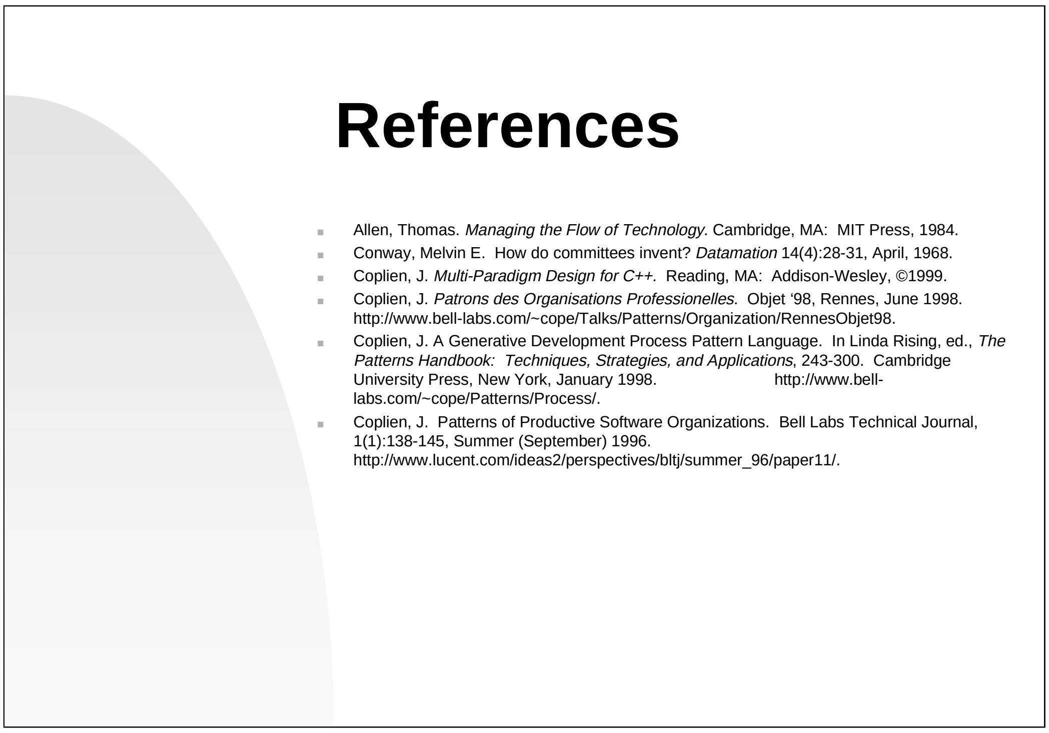


Conclusion

- Architecture and design are really about organizational structure
- Ownership of artifacts, system relationships
- A singular opportunity for curriculum developers, change agents, technologists, and corporate strategists

Organizations have architecture, too — in fact, that's the important architecture of a system. The software architecture is kind of incidental. Software architecture is a second-order consideration; it's the people that are primary. It's critical that this perspective permeate our curricula and management policies more universally.

Have you done something good for your fellow programmer today?



References

- Allen, Thomas. *Managing the Flow of Technology.* Cambridge, MA: MIT Press, 1984.
- Conway, Melvin E. How do committees invent? *Datamation* 14(4):28-31, April, 1968.
- Coplien, J. *Multi-Paradigm Design for C++.* Reading, MA: Addison-Wesley, ©1999.
- Coplien, J. *Patrons des Organisations Professionelles.* Objet '98, Rennes, June 1998. http://www.bell-labs.com/~cope/Talks/Patterns/Organization/RennesObjet98.
- Coplien, J. A Generative Development Process Pattern Language. In Linda Rising, ed., *The Patterns Handbook: Techniques, Strategies, and Applications*, 243-300. Cambridge University Press, New York, January 1998. http://www.bell-labs.com/~cope/Patterns/Process/.
- Coplien, J. Patterns of Productive Software Organizations. Bell Labs Technical Journal, 1(1):138-145, Summer (September) 1996. http://www.lucent.com/ideas2/perspectives/bltj/summer_96/paper11/.