AT&T Bell Laboratories

A Development Process Generative Pattern Language

James O. Coplien – AT&T Bell Laboratories

(708) 713-5384

cope@research.att.com

1. Introduction

This paper introduces a family of patterns that can be used to shape a new organization and its development processes. Patterns support emerging techniques in the software design community, where they are finding a new home as a way of understanding and creating computer programs. There is an increasing awareness that new program structuring techniques must be supported by suitable management techniques, and by appropriate organization structures; organizational patterns are one powerful way to capture these.

We believe that patterns are particularly suitable to organizational construction and evolution. Patterns form the basis of much of modern cultural anthropology: a culture is defined by its patterns of relationships. Also, while the works of Christopher Alexander[1] deal with town planning and building architecture to support human enterprise and interaction, it can be said that organization is the modern analogue to architecture in contemporary professional organizations. Organizational patterns have a first-order effect on the ability of people to carry on. We believe that the physical architecture of the buildings supporting such work are the dual of the organizational patterns; these two worlds cross in the work of Thomas Allen at MIT.[2]

There is nothing new in taking a pattern perspective to organizational analysis. What is novel about the work here is its attempt to use patterns in a generative way. All architecture fundamentally concerns itself with control;[3] here we use architecture to supplant process as the (indirect) means to controlling people in an organization. Not only should patterns help us understand existing organizations, but they should help us build new ones. A good set of organizational patterns helps to (indirectly) generate the right process: this indirectness is the essence of Alexandrine generativity. In fact, organizational patterns might be the most generative approach to software architecture patterns. Alexander notes that architectures "can't be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed."[4] As with many of the principles in *Timeless Way*, this is curiously reminiscent of the Way of Non-Action (*Yin*) of the Tao Teh Ching; we also find it in the triple-loop thinking of Swieringa and Wierdsma's organizational learning model.[5]

At this writing, the work is speculative: only limited use has been made of these patterns in formulating new organizations. The "goodness" or "badness" of such patterns is difficult to test by experiment. First, any metric of organiza-

---

1. Alexander, Christopher. *The Timeless Way of Building*. New York: Oxford University Press, 1979.
2. Allen, Thomas. *Managing the Flow of Technology.* Boston: MIT Press, ©1977, 141-182.
3. Interview with Jamee Carlin, architect, 1 July 1994.
4. Alexander, op. cit., p. xi.
5. Swieringa, Joop, and André Wierdsma. *Becoming a Learning Organization*. Reading, Mass.: Addison-Wesley, ©1992.

tional goodness is necessarily multidimensional and complex. Second, it is difficult to do large-scale social experiments with tight enough control variables that the effectiveness of a pattern could be verified. Third, such an experiment would take a long-term commitment (months or years), more than most software organizations are willing to expend in light of fragile and evolving markets.

For these reasons, the patterns fall back on empirical studies and on common sense. We look at recurring patterns of interaction in organizations, note recurring patterns between those patterns and some measures of “goodness,” and then do analysis to explain the correlation. The patterns presented here all combine empirical observations with a rationale that explains them. The claim that the language as a whole captures essential characteristics of high-productivity organizations has been validated by the CEOs of many small, highly productive organizations who have read these patterns.

The patterns meet other “standard” criteria emerging in the patterns movement. Each is stated as a problem or opportunity. Each is analyzed for the forces at play within it. Each instructs us to do something explicit. This form follows the pattern work of Alexander, whose books on architecture serve as a model for the modern software patterns movement. Some of these organizational patterns hark back to Alexander; where appropriate, a distinctive reference appears in the text (§*pattern-number*) to refer the reader to a pattern in Alexander’s *A Pattern Language*.[6]

Though there are organizations that exhibit these patterns, combining them into a new organization, built from scratch, is a daunting task. The ideal organization envisioned by these patterns differs greatly from the state of the art in software development. These patterns are drawn from peculiar organizations with peculiarly high productivity. The patterns describe practices much different from those found in most project management texts.

1.1 Language Context

This paper presents generative patterns that can be used to build an organization and to guide its development process in the domain of software development. By “organization”, I mean not so much an institutional organization as one of the communities of mutual interest that form naturally in any culture. These are sometimes called *instructive organizations*.[7] An organization is usually responsible for some deliverable. The endeavor of building that product within that organization is called a *project*. The patterns of activity within an organization (and hence within its project) are called a *process*. Organization, project, and process can be viewed as partial facets of each other. We might find hierarchies of organizations; if so, the patterns here apply in particular to the innermost ones, and in lesser degrees to the more encompassing organizations (see *Divide and Conquer* on page 27).

I am not so much interested in small (read “simple”) projects as in ambitious, complex commercial endeavors that may comprise hundreds of thousands or millions of lines of code. Such projects are common in telecommunications development, and their processes and organizations are a challenge to design. These organizations range in size from a handful of people to a few dozen people. Larger organizations (of hundreds or thousands of people) are beyond the scope of most of the patterns in this language. If such large organizations can be broken into smaller, *decoupled* organizations, the patterns apply to each of the parts.

All development organizations serve a purpose. This paper does not explore patterns for chartering a development organization, but presumes that an organization is formed within the industry, or within a company, to meet a business need economically. Questions of business practicality are beyond what can be dealt with in depth here. One might look at “Web of Shopping” (§19), “Household Mix” (§35) in Alexander as analogies to how such groups should be formed. Even grosser patterns such as “Site Repair” (§104) provide analogies for finding corporate contexts suitable to high-productivity organizations.

6. Alexander, Christopher, et al. *A Pattern Language*. New York: Oxford University Press, ©1977.
7. Swieringa and Wierdsma, *op. cit*.

More needs to be added, both here and to the pattern language, about the context for starting up an organization. How well-defined is the product at the outset? How much external tool support is available? We also need to study in more depth the degree to which these are, or are not, drivers for successful organizations.

1.2 Forces Driving the Language

While the language encodes well-known and reasonable folklore and practice about organizations, it also draws on unconventional insights gained through empirical studies of outstanding organizations. This pattern language builds on three years of research in development process and organizational analyses carried out by AT&T Bell Laboratories. The data come both from within AT&T and from other companies in computer-related fields. During this research, we used a largely visual representation of the organizations we studied, and built a catalogue of "*Gamma patterns*" of organizations, their structures, and their processes. These Gamma pattern visualizations are the source of many of the recurring *generative* patterns that we encode here. For example, the pattern *Engage QA* appears as a tight coupling between the QA role and the process as a whole in many highly productive organizations we studied. The original visualizations for these organizations are shown along with the pattern *Developer Controls Process* on page 11 and with the pattern *Engage QA* on page 16.

This language builds a system which is part of a larger system: a development culture in a corporate environment. Executing this language perfectly is no guarantee of success. While the language attempts to deal with interfaces to marketing and to the corporate control structure (e.g., through *Firewalls* on page 20), the remainder of the organization must be competent. Other patterns are needed to shape the remainder of the organization. We find hints of these patterns in the pattern languages of Kerth[8] and Whitenack.[9]

This language certainly works best when the raw materials are available to build an enthusiastic organization. While the patterns work better when building an organization from scratch, they may be applied to an anemic organization to restore the ability to excel that people may once have had, before the culture drained that ability away. Some "Site Repair" (§104) would likely be necessary before these patterns could be applied to a legacy organization. Some patterns are best-suited to the problems of mature organizations that experience difficulties (*Shaping Circulation Realms* on page 21; *Divide and Conquer* on page 27; *Hub-Spoke-and-Rim* on page 28; *3 to 7 Helpers per Role* on page 26).

1.3 The Quality Without a Name in Organizations

Some organizations have the "quality without a name:" they are true to their purpose because they profit their stockholders, serve their customers well, and provide a sustaining, fulfilling, and supporting workplace for their employees. Organizations that reach this level of maturity are likely to excel in all other measures. Do the patterns presented here make it possible to construct such an organization? While these patterns define a culture that helps sustain morale and business success, they don't address quality of work life and morale *directly*. These patterns touch only lightly on key factors such as the mutual trust between worker and peer, employee and employer, and company and customer. Other factors, such as cultural diversity, the corporation's community involvement, and extracurricular activities at work are beyond the scope of this language. I don't omit these factors here because I feel they are unimportant, but defer these issues to experts in organizational psychology and sociology.[10] Though many of the patterns here have sociological overtones, and likely have a generative effect on morale and other social indicators, other pattern languages must be woven with this one to round out the sociological picture. This perspective leads to the obvious conclusion that these patterns alone do not guarantee success, but it also provides a guiding light for further research.

8. Kerth, N. "The Caterpillar's Fate." Proceedings of PLoP/94.
9. Whitenack, Bruce. "RAPEL: A Requirements Analysis Pattern Language for Object-Oriented Development." Proceedings of PLoP/94.
10. Katz, Daniel, and Robert L. Kahn. *The Social Psychology of Organizations*, 2d ed. ©1978, John Wiley and Sons.

1.4 Language Rationale

This language takes inspiration as much from Alexander's architectural language as from his pattern language principles. The opening phrases of the language evoke the same sizing and context perspectives as we find in Alexander. Many of the organization patterns are refinements of Alexander's Circulation patterns (§98). The philosophy of establishing stable communication paths across the industry has strong analogies with the Alexandrian patterns that establish transportation webs in a city (§16). Here, we are concerned with "transportation" of information between individuals and development groups. We capture the communication between *roles*, which are a higher-order abstraction than individual actors.

1.5 Notations

This work draws on data gathered as part of the Pasteur process research program at AT&T Bell Laboratories.[11] Pasteur studied software development organizations in many companies worldwide, covering a wide spectrum of development cultures. The Pasteur analysis techniques are based in part on organizational visualization. Many of the patterns in this pattern language have visual analogues in the Pasteur analyses. I sometimes use visualizations to illustrate a pattern.

There are two kinds of pictures used in the Pasteur studies. The first is a social network diagram, also called an adjacency diagram. Each diagram is a network of roles and the communication paths between them. The roles are placed according to their coupling relationships: closely coupling roles are close together, and de-coupled roles are far apart. Roles at the center of these pictures tend to be the most active roles in these organizations, while those nearer the edges have a more distant relationship with the organization as a whole.

The second kind of picture is an interaction grid. The axes of the interaction grid span the roles in the organization, ordered according to their coupling to the organization as a whole. If a role at ordinate position p initiates an interaction with a role at coordinate position q, we put a point at the position (p,q). The point is shaded according to the strength of the interaction.

The pattern texts, and particularly the design rationales, often make reference to documents or projects that typify the pattern. "QPW" refers to Borland's QuattroPro for Windows development, and on process research conducted there by AT&T Bell Labs in 1993. The research is further discussed in the proceedings of BIC/94[12], in a column by Richard Gabriel[13], and in an article in Dr. Dobb's journal.[14]

The notation "ATT1" refers to a high productivity project inside AT&T, characterized by concurrent engineering in a small team environment. Findings from research on that project have not yet been published.

The analyses occasionally make reference to the work of Thomas Allen, whose relevant perspectives can be found in his book on technology information flow.[15]

I often draw from a whimsical book on architecture called *The Most Beautiful House in the World*, by Witold Rybczynski.[16]

Cites to Alexander refer to *A Pattern Language*, and use paragraph designations (§) to refer to pattern numbers therein.

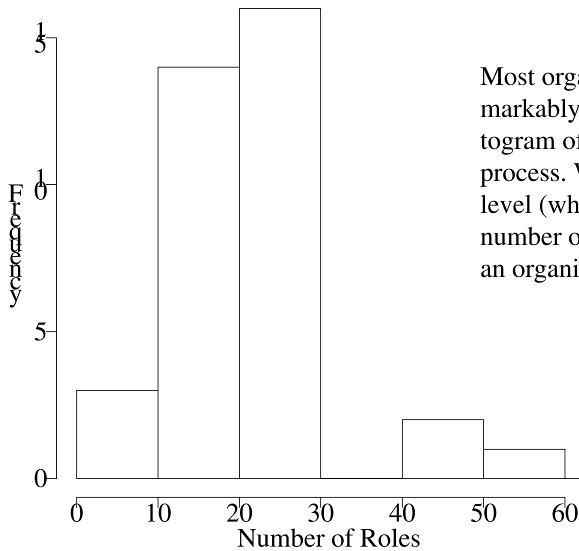
---

11. Cain, B. G., and J. O. Coplien. "A Role-based empirical process modeling environment." *Proceedings of the Second International Conference on Software Process*, Berlin, February 1993.
12. Coplien, J. O. "Borland Software Craftsmanship: A new look at process, quality, and productivity." *Proceedings of the 5th Annual Borland International Conference*, Orlando, FL, June 1994.
13. Gabriel, Richard P. "Productivity: Is there a silver bullet?" *Journal of Object-Oriented Programming*, Vol. 7, No. 1, March/April 1994, pp. 89-92.
14. Coplien, J. O. "Evaluating the Software Development Process." *Dr. Dobb's Journal,* Vol. 19, No. 11, October, 1994.
15. Allen, *op. cit.*
16. Rybczynski, Witold. *The Most Beautiful House in the World*. New York: Penguin, ©1989.

1.6 Acknowledgments

Brendan Cain supported much of the early process research underlying these patterns.[17] The research of Peter Bürgi, who spent time as a cultural anthropologist in our Research organization, has strongly influenced the pattern language. The pattern language took shape and had its first applications in the formative months of the Global Configurator project at Indian Hill, whose many members I thank for the opportunity to work with them. Joint work with Neil Harrison of AT&T's Global Business Communications Systems helped refine, validate, and prove in many of the patterns. Many thanks to Richard Gabriel, Ward Cunningham, Desmond DeSouza, Richard Helm and Ralph Johnson, for participating in a structured review of these patterns; Gabriel also facilitated the review at PLoP. Many thanks to the many people at PLoP who also reviewed these patterns, and especially to Larry Podmolik, Mary Shaw, Dennis DeBruler, Norm Kerth, Bill Opdyke, Brian Foote, Bruce Whitenack, Steve Berczuk, Frank Buschmann, and Robert Martin. Kent Beck was the PLoP shepherd for this document. Mary Zajac provided much useful feedback from a management perspective. Many comments and pattern ideas, most noteworthy of which is Mercenary Analyst, came from Paul Chisholm at AT&T.

17. Cain and Coplien, op. cit.



Most organizations we have studied have a remarkably similar number of roles. This is a histogram of the distribution of roles (nodes) per process. While this is not the same as the staff level (which is the number of actors, not the number of roles) it provides guidance in sizing an organization.

---

2. Pattern Name: Size the Organization

Problem: How big should the organization be?

Context:

You are building a software development organization to meet competitive cost and schedule benchmarks. The first release of the end product will probably more than 25,000 lines of code.

The development organization is embedded in the context of a larger organization, usually that of the sponsoring enterprise or company (see the ***Rationale:***).

Forces:

Too large, and you reach a point of greatly diminishing returns.

Large organizations are ships that are hard to steer.

Too small, and you don't have critical mass.

Overly small organizations have inadequate inertia and can become unstable.

Size affects the deliverable non-linearly. Communication overhead goes up as the square of the size, which means that the organization becomes less cohesive as the square of the size while the "horsepower" of the organization goes up only linearly.

New people interact much more strongly with others in the organization, and interact more broadly across the organization, than mature members of the society who have assimilated its context.

Solution:

By default, choose 10 people; experience has shown that suitably selected and nurtured small team can develop a 1,500 KSLOC project in 31 months, and a 200 KSLOC project in 15 months, or a 60KSLOC project in 8 months. Do not add people late in development, or try to meet deadlines worked backward from a completion date.

Resulting Context:

An organization where nearly everyone can have "global knowledge." We have found empirically that most roles in a project can handle interactions with about six or seven other roles; with 10 people, you can almost manage total global communications (and a fully connected network may not be necessary). On the other hand, 10 people is a suitable "critical mass."

This pattern interacts with *3 to 7 Helpers per Role* on page 26.

Design Rationale:

Keeping an organization small makes it possible for everybody to know how the project works. Projects that do well have processes that adapt, and processes adapt well only if there is widespread buy-in and benefit. The dialogue necessary to buy-in and benefit can accrue only to small organizations. Tom DeMarco has noted that everybody who is to benefit from process should be involved in process work and process decision-making.[18]

Having 10 people at the start is probably overkill, but it avoids the expense and overhead of adding more people later.

This pattern is for large projects. Projects larger than 25KSLOC can rarely be done by an individual (see *Solo Virtuoso* on page 7).

Different management styles (leadership-based, manager-based) lead to the success or failures of different organization types (democracy, republic, oligarchy). Leadership-based management styles help these organizations work; the compelling need for a democracy is less felt, because project members feel they are well-represented under an appropriate management style. [This may be a new pattern about management style.]

A single team is better than a collection of sub-teams. The faster a team breaks up into sub-teams worrying about their own responsibilities rather than those of the larger team, the less effective the enterprise will be as a whole.

Further study might evaluate the relationship between this pattern and Alexander's "The Distribution of Towns" (§2) and related patterns. Here, we stipulate that the social organization must be small; it reflects a "Subculture Boundary" (§13) and "Identifiable Neighborhood" (§14). Alexander emphasizes the grander architectural context that balances support for the ecology with the economies of scale that large towns can provide, while supporting the xenophobic tendencies of human nature. Small organizations like that being built here rarely exist in isolation, but in the context of a broader supporting organization. This relationship to the larger organization invokes *Patron* on page 12.

3. Pattern Name: Solo Virtuoso

Problem: How big should the organization be?

Context:

You are building a software development organization to meet competitive cost and schedule benchmarks. The first release of the end product will probably be less than 25 KSLOC.

Rapid growth is not anticipated after the first release.

Forces:

Some select individuals are able to build entire projects by themselves.

Organization size affects the deliverable non-linearly. Communication overhead goes up as the square of the size, which means that the organization becomes less cohesive as the square of the size while the "horsepower" of the organization goes up only linearly.

Solution:

Do the entire design and implementation with one or two people.

Resulting Context:

An organization limited to small developments. Though there is a singleton development role, other roles may be necessary to support marketing, toolsmithing, and other functions. The productivity of a suitably chosen singleton developer is enough to handle sizable projects; here, we establish 25KSLOC as a limit, but see the rationale below for further parameters.

This approach is rarely applicable, as it doesn't resolve some of the forces (many of them applicable) mentioned in *Size the Organization* on page 6. Where those forces don't apply, this pattern is a big win.

Design Rationale:

There are numerous examples of successful single-person developments. The dynamics of this development are different from those for a small team. The productivity of a single individual can be higher than a collection of several productive individuals. We have seen single-person developments generate 25KSLOC of deliver-

18. Personal discussion with Tom DeMarco, January, 1993.

able code in 4 months (a craft interface for a telecommunication system); two-person developments do 135 KSLOC in 30 months.

Success, of course, depends on choosing the right person. Boehm notes a 20-fold spread between the least and most effective developers. A telecommunications developer recently told me that “having the right expertise means the difference between being able to solve a problem in a half hour, and never being able to solve the problem at all.”

This pattern reduces the “truck number” of the organization (the number of people whose absence would threaten the organization were any one of them hit by a truck).

4. Pattern Name: Size the Schedule

Problem: How long should the project take?

Context:

The product is understood and the project size has been estimated.

Forces:

If you make the schedule too generous, developers become complacent, and you miss market windows.

If the schedule is too ambitious, developers become burned out, and you miss market windows.

If the schedule is too ambitious, product quality suffers, and compromised architectural principles establish a poor foundation for future maintenance.

Projects without schedule motivation tend to go on forever, or spend too much time polishing details that are either irrelevant or don’t serve customer needs.

Solution:

Reward developers for meeting the schedule, with financial bonuses (or at-risk compensation), or with extra time off. Keep two sets of schedules: one for the market, and one for the developers. The external schedule is negotiated with the customer; the internal schedule, with development staff. The internal schedule should be shorter than the external schedule by two or three weeks for a moderate project (this figure comes from a senior staff member at a well-known software consulting firm). If the two schedules can’t be reconciled, customer needs or the organization’s resources—or the schedule itself—must be re-negotiated.

Resulting Context:

A project with a flexible target date. Dates are always difficult to estimate; DeMarco notes that one of the most serious signs of a project in trouble is a schedule worked backward from an end date.[19]

Keep the schedule current with *Take No Small Slips* on page 31; see also *Interrupts Unjam Blocking* on page 32.

Other mechanisms are necessary to ensure that a hurried development doesn’t compromise quality.

Design Rationale:

MIT project management simulation; QPW.

5. Pattern Name: Form Follows Function

Alias: Aggregate Roles into Activities

Problem: A project lacks well-defined roles.

Context:

You know the key atomic process activities.

Forces:

Activities are too small, and their relationship too dynamic, to be useful process building blocks.

Activities often cluster together by related artifacts or other domain relationships.

---

19. Tom DeMarco, CaseWorld Conference, January, 1993.

Solution:

Group closely related activities (that is, those mutually coupled in their implementation, or which manipulate the same artifacts, or that are semantically related to the same domain). Name the abstractions resulting from the grouped activities, making them into roles. The associated activities become the responsibilities (job description) of the roles.

Resulting Context:

A *partial* definition of roles for a project. Some roles (mercenary analyst, developer) are canonical, rather than deriving from this pattern.

Design Rationale:

The quality of this pattern needs to be reviewed. The idea came from the approach taken in a large project re-engineering effort I worked with in March of 1994.

Louis Sullivan is the architect credited with the primordial architectural pattern of this name (q.v. Rybczynski, *The Most Beautiful House in the World*, p. 162).

This pattern interacts with other structural patterns such as *Organization Follows Location* on page 9, *Organization Follows Market* on page 10, and *Architect Controls Product* on page 13. See, in particular, *Engage Customer* on page 17.

6. Pattern Name: Domain Expertise in Roles

Problem: Matching staff (actors, people) to roles.

Context:

You know the key atomic process roles, including a characterization of the *Developer* role.

Forces:

All roles must be staffed.

All roles must be staffed with qualified individuals.

Spreading expertise across roles complicates communication patterns.

Solution:

Hire domain experts. Any given actor may fill several roles. In many cases, multiple actors can fill a given role. Domain training is more important than process training. Local gurus are good, in all areas from application expertise to expertise in methods and language.

Resulting Context:

This is a tool that helps assure that roles can be successfully carried out. It also helps make roles autonomous.

Other roles (*Architect*, *Mercenary Analyst*, and others) are prescribed by subsequent patterns.

Design Rationale:

Highly productive projects (e.g., QPW) hire deeply specialized experts.

Alexander §40, “Old People Everywhere,” talks about the need of the young to interact with the old. The same deep rational and many of the same forces of Alexander’s pattern also apply here.

7. Pattern Name: Organization Follows Location

Problem: Assigning tasks and roles across a geographically distributed work force.

Context:

A product must be developed in several different hallways, on different floors of a building, in different buildings or at different locations.

Forces:

Communication patterns between project members follows geographic distribution.

Coupling between pieces of software must be sustained by analogous coupling between the people maintaining that software.

People avoid communicating with people who work in other buildings, other towns, or overseas (see the ***Rationale***).

People in an organization usually work on related tasks, which suggests that they communicate frequently with each other.

Solution:

The architectural partitioning should reflect the geographic partitioning, and vice versa. Architectural responsibilities should be assigned so decisions can be made (geographically) locally.

Resulting Context:

Sub-organizations that can be further split or organized by market or other criteria (see *Organization Follows Market* on page 10, *Shaping Circulation Realms* on page 21, and others).

You still need someone to break logjams when consensus can't be reached, perhaps using *Architect Controls Product* on page 13.

Design Rationale:

Thomas Allen has found that social distance goes up rapidly with physical separation (see also "House Cluster" (§37) of Alexander). Our empirical experience with co-development projects overseas reveals that failure to follow this pattern can lead to complete project failure. This is a crucial pattern that is often overlooked or dismissed out of consideration for political alliances. Peter Bürgi's studies of geographically distributed organizations in AT&T underscore the importance of this pattern.

We have seen few geographically distributed organizations that exhibit positive team dynamics. There are exceptions, and there are rare occasions when this pattern does not apply. Steve Berczuk at MIT notes: "... communications need not be poor between remote sites if the following items are true: 1) the number of developers on a project, including all sites is small; 2) most of the communication is done via something like email (wide distribution and asynchronous communication—in [one case of his experience] ... more people were in the loop than if the primary means of communication had been hallway chats); 3) The people involved have been together for SOME time so that they feel like they know each other; 4) Folks aren't so burned out by 'unnecessary' travel that they are willing and happy to travel when it is needed. In some situations pattern 7 is not possible because of the nature of the project, so we need a way to address the issue of remoteness." (Personal communication with Steve Berczuk, August, 1994.)

8. Pattern Name: Organization Follows Market

Problem: There is no clear role or organizational accountability to individual market segments.

Context:

The market comprises several customers with similar but conflicting needs.

The project has adopted sound architectural principles, and can organize its software according to market needs.

Forces:

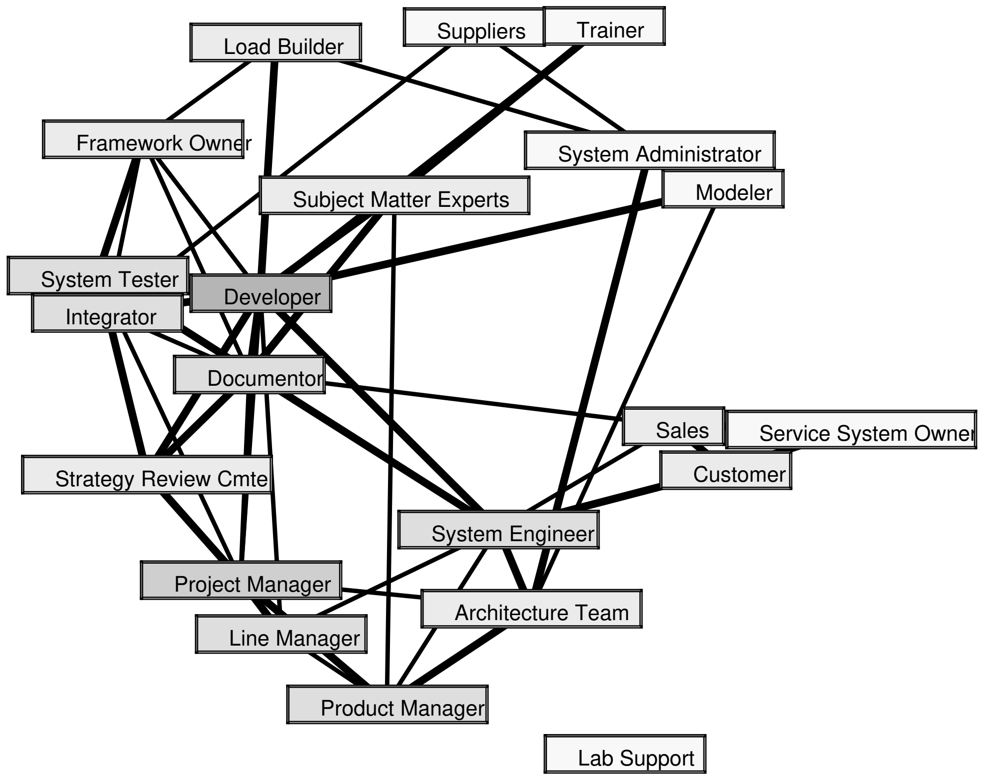
The development organization should track and meet the needs of each customer.

Customer needs are similar, and much of what they all need can be done in common.

Different customers expect results on different schedules.

Solution:

In an organization designed to serve several distinct markets, it is important to reflect the market structure in the development organization. One frequently overlooked opportunity for a powerful pattern is the conscious design of a "core" organization, that supports only what is common across all market segments. Ralph Johnson calls this a *framework team*. It is important to put this organization in place up front.



The developer is central to all activities of this end-to-end software development process (see Pattern 9)

Resulting Context:

An organization that can support a good architecture. The success of this pattern is necessary to the success of *Architect Controls Product* on page 13. *Architect Controls Product* on page 13 should be seen as an audit, refinement, or fine-tuning of this pattern.

Design Rationale:

Most of the rationale is in the forces. Two of the major forces relate to individual customer schedules, and to posture the organization to respond quickly to customer requests. Two important aspects of domain analysis are broadening the architecture (e.g., by working at the base class level), and ensuring that architectural evolution tracks the vendor understanding of customer needs. A single organization can't faithfully track multiple customer needs, and this organization allows different arms of the organization to track different markets independently.

9. Pattern Name: Developer Controls Process

Problem: What role should control communication across the process?

Context:

An imperfectly understood design domain, where iteration is key to development.

Forces:

Totalitarian control is viewed by most development teams as a draconian measure.

The right information must flow through the right roles.

You need to support information flow across analysis, design, and implementation.

Managers have some accountability.

Developers should have ultimate accountability, and have the authority and control of the product; these are often process issues.

Solution:

Place the *Developer* role at a hub of the process for a given feature. A feature is a unit of system functionality (implemented largely in software) that can be separately marketed, and for which customers are willing to pay. The *Developer* is the process information clearinghouse. Responsibilities of *Developers* include understanding requirements, reviewing the solution structure and algorithm with peers, building the implementation, and unit testing.

Note that other hubs may exist as well.

Resulting Context:

An organization that supports its prime information consumer.

The *Developer* can be moved toward the center of the process using patterns *Shaping Circulation Realms* on page 21, and *Move Responsibilities* on page 21.

Though *Developer* should be a key role, care must be taken not to overburden it. This pattern should be balanced with *Mercenary Analyst* on page 19, *Firewalls* on page 20, *Firewalls* on page 20, and more general load-balancing patterns like *Buffalo Mountain* on page 23.

Design Rationale:

We have no role called *Designer* because design is really the whole task. *Manager*s fill a supporting role; empirically, they are rarely seen to control a process except during crises. While the *Developer* controls the process, the *Architect* controls the product. (In the figure, the *Architect* role is split across *Framework_Owner* and *Architecture Team*.) This communication is particularly important in domains that are not well understood, so that iteration can take place to explore the domain with the customer.

10. Pattern Name: Patron

Problem: Giving a project continuity.

Context:

A development organization where roles are being defined. *Patron* works only if *Developer Controls Process* on page 11 is in place.

Forces:

Centralized control can be a drag.

Anarchy can be a worse drag.

Most societies need a king/parent figure.

Solution:

Give the project access to a visible, high-level manager, who will champion the cause of the project.

Resulting Context:

Having a patron gives the organization a sense of being, and a focus for later process and organizational changes. Other roles can be defined in terms of the patron's role.

The manager role is not to be a totally centralized control, but rather a champion. That is, the scope of the manager's influence is largely outside those developing the product itself, but includes those whose cooperation is necessary for the success of the product (support organizations, funders, test organizations, etc.). This role also serves as a patron or sponsor; the person is often a corporate visionary.

Design Rationale:

We have observed this in Phillippe Kahn in QPW; Sethi et al. in C++ efforts in AT&T; for a manager in a high-productivity Network Systems project; and in another multi-location AT&T project.

This relates to the pattern *Firewalls* on page 20, which in turn relates to the pattern *Firewalls* on page 20.

Block talks about the importance of influencing forces over which the project has no direct control.[20]

"The term pattern comes from Middle English *patron* (and the more ancient French *patron*) which still means both 'patron' and 'pattern.' In the 16th century, patron, with a shifted accent, evidently began to be pronounced patrn, and spelt patarne, paterne, pattern. By 1700 the original form ceased to be used of things, and *patron*

20. Block, R. *Politics of Projects.* Yourdon Press: 1983.

and *pattern* became differentiated in form and sense. 1 a ‘The original proposed to imitation; the archetype; that which is to be copied; an exemplar’ (J.); an example or model deserving imitation; an example or model of a particular excellence. aC. 1369 CHAUCER Dethe Blaunche 910 Truely she Was her chefe patron of beaute, And chefe ensample of al her werke.” – *From a dictionary of medieval terms, related by Aamod Sane at University of Illinois*.

Putting the developer in charge of the process implies that management (see *Firewalls* on page 20) titles become associated with support roles. This works only in a culture where the manager *decides* to be the servant of the developer (an insight from Norm Kerth).

11. Pattern Name: Architect Controls Product

Problem: A product designed by many individuals lacks elegance and cohesiveness.

Context:

An organization of *Developers* that needs strategic technical direction.

Forces:

Totalitarian control is viewed by most development teams as a draconian measure.

The right information must flow through the right roles.

Solution:

Create an *Architect* role. The *Architect* role should advise and control *Developer* roles, and should communicate closely with them. The *Architect* should also be in close touch with *Customer*.

Resulting Context:

This does for the architecture what the *Patron* does for the organization: it provides technical focus, and a rallying point for technical work as well as market-related work.

There is a rich relationship between this pattern and *Patron* on page 12 that should be explored.

Design Rationale:

We have no role called *Designer* because design is really the whole task. Managers fill a supporting role; empirically, they are rarely seen to control a process except during crises. While the *Developer* controls the process, the *Architect* controls the product. The *Architect* is a “chief *Developer*” (see pattern *Architect Controls Product* on page 13). Their responsibilities include understanding requirements, framing the major system structure, and controlling the long-term evolution of that structure.

The *Architect* controls the product in the visualization accompanying the pattern *Engage QA* on page 16.

“Les oeuvres d’un seul architect sont plus belles...que ceux d’ont plusiers ont taché de faire.” – Pascal, *Pensées*.

12. Pattern Name: Conway’s Law

Alias: Organization follows architecture

Problem: Aligning organization and architecture

Context:

An *Architect* and development team are in place.

Forces:

Architecture shapes the communication paths in an organization.

De facto organization structure shapes formal organization structure.

Formal organization structure shapes architecture.

Solution:

Make sure the organization is compatible with the product architecture. At this point in the language, it is more likely that the architecture should drive the organization than vice versa.

Resulting Context:

The organization and product architecture will be aligned.

Design Rationale:

Historical.

13. Pattern Name: Architect Also Implements (αρχι_τεκτον)

Problem: Preserving the architectural vision through to implementation

Context:

An organization of *Developers* that needs strategic technical direction.

Forces:

Totalitarian control is viewed by most development teams as a draconian measure.
The right information must flow through the right roles.

Solution:

Beyond advising and communicating with *Developers, Architects* should also participate in implementation.

Resulting Context:

A development organization that perceives buy-in from the guiding architects, and that can directly avail itself of architectural expertise.

Design Rationale:

The importance of making this pattern explicit arose recently in a project I work with. The architecture team was being assembled across wide geographic boundaries with narrow communication bandwidth between them. Though general architectural responsibilities were identified and the roles were staffed, one group had expectations that architects would also implement code; the other did not.

"It would be convenient if architecture could be defined as any building designed by an architect. But who is an architect? Although the Academie Royale d'Architecture in Paris was founded in 1671, formal architectural schooling did not appear until the nineteenth century. The famous Ecole des Beaux-Arts was founded in 1816; the first English-language school, in London, in 1847; and the first North American university program, at MIT, was established in 1868. Despite the existence of professional schools, for a long time the relationship between schooling and practice remained ambiguous. It is still possible to become an architect without a university degree, and in some countries, such as Switzerland, trained architects have no legal monopoly over construction. This is hardly surprising. For centuries, the difference between master masons, journeymen builders, joiners, dilettantes, gifted amateurs, and architects has been ill defined. The great Renaissance buildings, for example, were designed by a variety of non-architects. Brunelleschi was trained as a goldsmith; Michelango as a sculptor, Leonardo da Vinci as a painter, and Alberti as a lawyer; only Bramante, who was also a painter, had formally studied building. These men are termed architects because, among other things, they created architecture–a tautology that explains nothing." – Witold Rybczynski, *The Most Beautiful House in the World*.

14. Pattern Name: Review

Problem: Blind spots in the architecture and design

Context:

A software artifact whose quality is to be assessed for improvement.

Forces:

Architecture decisions affect many people over a long time.
Individual *Architects* and *Designers* can develop tunnel vision.
A shared architectural vision is important.
Even low-level design and implementation decisions matter.
All things are deeply interwingled (Ed Yourdon).

Solution:

All architectural decisions should be reviewed by all *Architects*. *Architects* should review each others' *code*. The reviews should be frequent–even daily–early in the project. Reviews should be informal, with a minimum of paperwork.

Resulting Context:

This pattern sets the context for *Mercenary Analyst* on page 19. It will also solve potential problems that have been pointed out for *Code Ownership* on page 15.

The intent of this pattern is to increase coupling between those with a stake in the architecture and implementation, which solves the stated problem indirectly.

Design Rationale:

QPW; a successful object-oriented project in AT&T.

15. Pattern Name: Code Ownership

Problem: A *Developer* cannot keep up with a constantly changing base of implementation code.

Context:

A system with mechanisms to document and enforce the software architecture, and developers to write the code.

Forces:

Something that's everybody's responsibility is no one's responsibility.
You want parallelism between developers, so multiple people can be coding concurrently.
Most design knowledge lives in the code; navigating unfamiliar code to explore design issues takes time.
Provisional changes never work.
Not everyone can know everything all the time.

Solution:

Each code module in the system is owned by a single *Developer*. Except in exceptional and explicit circumstances, code may be modified only by its owner.

Resulting Context:

The architecture and organization will better reflect each other. Related patterns include *Architect Controls Product* on page 13, *Organization Follows Market* on page 10, and *Interrupts Unjam Blocking* on page 32.

The pattern *Review* on page 14 helps keep *Designers* and *Architects* from developing tunnel vision from strict application of this pattern.

Design Rationale:

Lack of code ownership is a major contributor to discovery effort in large-scale software development today. Note that this goes hand-in-hand with architecture: to have ownership, there must be interfaces. This is a form of *Conway's-law*-in-the-small (see *Architect Controls Product* on page 13 and *Architect Controls Product* on page 13).

Arguments against code ownership have been many, but empirical trends uphold their value. Typical concerns include the tendency toward tunnel vision, the implied risk of having only a single individual who understands a given piece of code in-depth, and breakdown of global knowledge. Other patterns temper these problems (see *Review* on page 14 above).

Tim Born argues that there is a relationship between code ownership and encapsulation, in the sense that C++ protection keeps one person from accessing the implementation of another's abstraction.

Law is property, and the lack of identifiable property leads to anarchy (Rousseau et al.).

It has been argued that code ownership should be applied only to reusable code. Such a constraint would be worthy of consideration if someone comes up with a good distinction between usable code and reusable code.

16. Pattern Name: Application Design is Bounded By Test Design

Problem: When do you design and implement test plans and scripts?

Context:

A system with mechanisms to document and enforce the software architecture, and developers to write the code. A *Testing* role is being defined.

Forces:

Test development takes time, and cannot be started just when the system is done ("when we know what we have to test").

Scenarios are known when requirements are known, and many of these are known early.

Test implementation needs to know the details of message formats, interfaces, and other architectural properties in great details (to support test scripts and test jigs).

Implementation changes daily; there should be no need for test designs to track ephemeral changes in software implementation.

Solution:

Scenario-driven test design starts when scenario requirements are first agreed to by the customer. Test design evolves along with software design, but only in response to customer scenario changes: the source software is inaccessible to the tester. When development decides that architectural interfaces have stabilized, low-level test design and implementation can proceed.

Resulting Context:

This provides a context for *Engage QA* on page 16 and for *Scenarios Define Problem* on page 18.

Design Rationale:

Making the software accessible to the tester causes them to see the developer view rather than the customer view, and leads to the chance they may test the wrong things, or at the wrong level of detail. Furthermore, the software will continue to evolve from requirements until the architecture gels, and there is no sense in causing test design to fishtail until interfaces settle down.

In short, test design kicks off at the end of the first major influx of requirements, and touches base with design again when the architecture is stable.

This is related to the (yet unspecified) pattern, "Testing first in last out," to the pattern *Engage QA* on page 16, and to the pattern *Scenarios Define Problem* on page 18.

17. Pattern Name: Engage QA

Problem: How do you guarantee product quality?

Context:

A set of roles in a development organization, and a customer, with a need for some filter between them to ensure the quality of the product.

Forces:

Developers feel they get everything right.

Perfect software is hard.

Quality is too often deferred.

Success depends on high quality.

Early feedback is important for fundamental quality problems.

Solution:

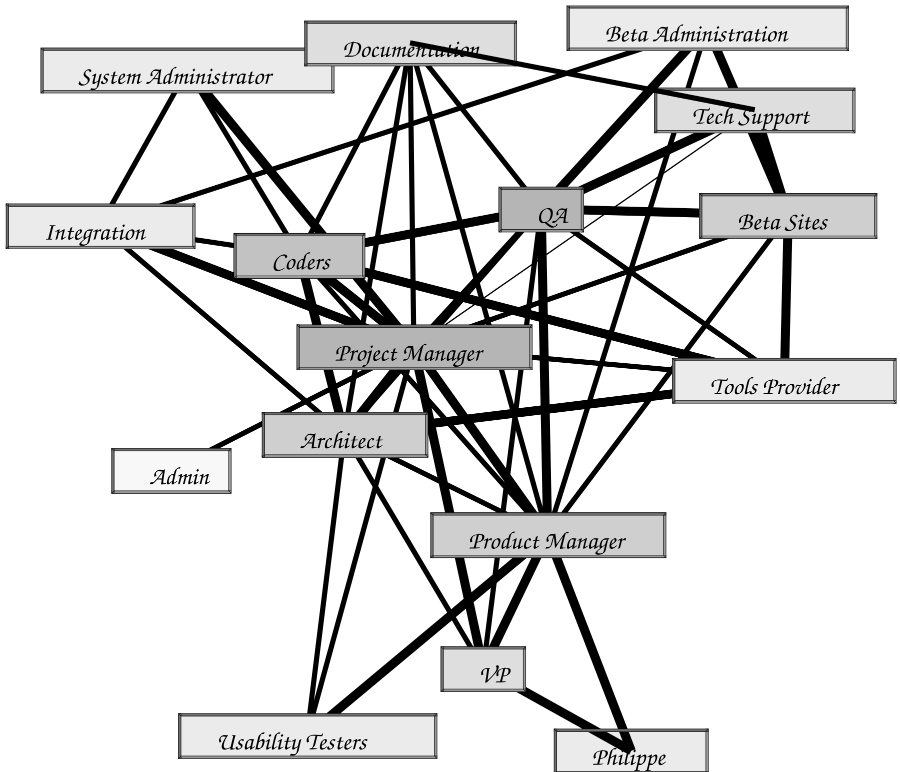
Make *QA* a central role. Couple it tightly to development as soon as development has something to test. Test plan development can proceed in parallel with coding, but *Developers* declare the system ready for test.

Resulting Context:

Having engaged *QA*, the project will be ready to approach the *Customer*. With *QA* and the *Customer* engaged, the quality process can be put in place (use cases gathered, etc.)

Design Rationale:

QPW. See also *Application Design is Bounded By Test Design* on page 15.



Quality Assurance (QA) was central to the development of Borland's Quattro Pro for Windows (see Pattern 17)

---

18. Pattern Name: Engage Customer

Problem: Maintaining customer satisfaction

Context:

A quality assurance function exists, and needs input to drive its work.

Forces:

Developers used to be called "loose cannons on deck."

Requirements changes occur even after design reviews are complete and coding as started.

Missing customer requirements is a serious problem.

Customers are traditionally not part of the mainstream development, which makes it difficult to discover and incorporate their insights.

Trust relationship between managers and coders.

Solution:

Make *Customer* a role that is closely coupled to the *Developer* and *Architect*, not just to *QA*.

Resulting Context:

The new context supports requirements discovery from the customer, as required by the pattern *Scenarios Define Problem* on page 18, and the pattern *3 to 7 Helpers per Role* on page 26. Other patterns like *Firewalls* on page 20 and *Firewalls* on page 20 also build on this pattern.

Design Rationale:

QPW. Also, see *Software Development and Reality Construction*, Floyd et al., eds., Berlin: Springer-Verlag, 1992, and in particular the works of Reisin and Floyd therein.

The project must be careful to temper interactions between *Customer* and *Developer*, using the patterns mentioned in the Resulting Context.

Note that "maintaining product quality" is not the problem being solved here. Product quality is only one component of customer satisfaction. Studies have shown that customers leave one company for another when they feel they are being ignored (20% of the time), or because the attention they receive was rude or unhelpful (50% of the time). For customers having problems that cost over $100 to fix, and the company does not fix it, only 9% would buy again. 82% would do business with the company again if the problem was quickly resolved after they complained. (The source for the former pair is The Forum Corporation; for the latter pair, Traveler's Insurance Company.[21])

19. Pattern Name: Group Validation

Problem: Ensuring product quality.

Context:

The quality of analysis, design, or implementation is to be assessed.

Forces:

The job of *QA* is to assess quality.

QA usually assesses the quality of the end product, doing only black-box validation and verification.

A group setting brings many insights on product problems and opportunities.

Individuals may not have the insight necessary to discover the bug plaguing the system (this may be an issue of objectivity).

Solution:

Even before engaging *QA*, the development team—including *Customer*—can validate the design. Techniques such as CRC cards and group debugging help socialize and solve problems.

Resulting Context:

A process where the quality of the system is constantly brought into focus before the whole team. In the resulting context, problems will be resolved sooner. The cost of this pattern is the time expended in group design/code debugging sessions.

Design Rationale:

The CRC design technique has been found to be a great team-builder, and an ideal way to socialize designs. Studies of GBCS projects have found group debugging sessions to be unusually productive.

Bringing the customer into these sessions can be particularly helpful. The project must be careful to temper interactions between *Customer* and *Developer*, using the patterns mentioned in the Resulting Context.

There is an empirical research foundation for this pattern. See "An implementation of structured walk-throughs in teaching COBOL programming," CACM, Vol. 22, No. 6, June, 1979, which found that team debugging contributes to team learning and effectiveness. A contrary position can be found in G. J. Meyers, "A controlled experiment in program testing and code walkthroughs/inspections," CACM, Vol. 21, No. 9, September, 1978, though this study was limited to fault detection rates and did not evaluate the advantages of team learning.

20. Pattern Name: Scenarios Define Problem

Problem: Design documents are often ineffective as vehicles to communicate the customer vision of how the system should work.

---

21. Zuckerman, M. R., and Lewis J. Hatala. *Incredibly American*. Milwaukee: ASQC Quality Press, ©1992, pp. 81-83.

Context:

You want to engage the customer and need a mechanism to support other organizational alliances between customer and developers.

Forces:

There is a natural business distancing and mistrust between customers and developers.

Communication between developers and customers is crucial to the success of a system.

Solution:

Capture system functional requirements as use cases, a la Jacobson.

Resulting Context:

The problem is now defined, and the architecture can proceed in earnest.

Design Rationale:

Jacobson; also CACM Nov. '88 (v. 31, no. 11) pp 1268-1287, according to Ralph Johnson. Also Rubin and Goldberg, who take scenarios all the way to the front of the process, preceding even design. See also, "Formal Approach to Scenario Analysis," Pei Hsia, Jayaranan Samuel, Jerry Gao, and David Kung, IEEE Software, March, 1994, Vol. 11, No. 2, ff 33.

21. Pattern Name: Mercenary Analyst

Problem: Supporting a design notation, and the related project documentation, is too tedious a job for people directly contributing to product artifacts.

Context:

You are assembling the roles for the organization. The organization exists in a context where external reviewers, customers, and internal developers expect to use project documentation to understand the system architecture and its internal workings. (User documentation is considered separately).

Forces:

If developers do their own documentation, it hampers "real" work.

Documentation is often write-only.

Engineers often don't have good communication skills.

Architects can become victims of the elegance of their own drawings (see rationale).

Solution:

Hire a technical writer, proficient in the necessary domains, but without a stake in the design itself. This person will capture the design using a suitable notation, and will format and publish the design for reviews and for consumption by the organization itself.

Resulting Context:

The success of this pattern depends on finding a suitably skilled agent to fill the role of mercenary analyst. If the pattern succeeds, the new context defines a project whose progress can be reviewed (the pattern *Review* on page 14) and monitored by community experts outside the project.

Design Rationale:

QPW; many AT&T projects (a joint venture based in New Jersey, a formative organization in switching support, and others).

"Here is another liability: beautiful drawings can become ends in themselves. Often, if the drawing deceives, it is not only the viewer who is enchanted but also the maker, who is the victim of his own artifice. Alberti understood this danger and pointed out that architects should not try to imitate painters and produce lifelike drawings. The purpose of architectural drawings, according to him, was merely to illustrate the relationship of the various parts... Alberti understood, as many architects of today do not, that the rules of drawing and the rules of building are not one and the same, and mastery of the former does not ensure success in the latter." – Witold Rybczynski, *The Most Beautiful House in the World*, p. 121.

22. Pattern Name: Firewalls

Alias: Manager

Problem: Project implementors are often distracted by outsiders who feel a need to offer input and criticism.

Context:

An organization of developers has formed, in a corporate or social context where they are scrutinized by peers, funders, customers, and other “outsiders.”

Forces:

Isolationism doesn’t work: information flow is important.
Communication overhead goes up non-linearly with the number of external collaborators.
Many interruptions are noise.
Maturity and progress are more highly correlated with being in control than being controlled.

Solution:

Create a *Manager* role, who shields other development personnel from interaction with external roles. The responsibility of this role is “to keep the pests away.”

Resulting Context:

The new organization isolates developers from extraneous external interrupts. To avoid isolationism, this pattern must be tempered with others, such as *Engage Customer* on page 17 and *Firewalls* on page 20.

Gatekeeper and *Firewall* alone are insufficient to protect developers in an organization whose culture allows marketing to drive development schedules.

Design Rationale:

QPW, ATT1. See also the various works of Gerry Weinberg.
See also the pattern *Engage Customer* on page 17, which complements this pattern.
Gatekeeper is a pattern that facilitates effective flow of useful information; *Firewalls* restricts flow of detracting information.

23. Pattern Name: Gatekeeper

Problem: Balancing the need to communicate with typically introverted engineering personality types.

Context:

An organization of developers has formed, in a corporate or social context scrutinized by peers, funders, customers, and other “outsiders.”

Forces:

Isolationism doesn’t work: information flow is important.
Communication overhead goes up non-linearly with the number of external collaborators.
Many interruptions are noise.
Maturity and progress are more highly correlated with being in control than being controlled.

Solution:

One project member, a Type A personality, rises to the role of *Gatekeeper*. This person disseminates leading-edge and fringe information from outside the project to project members, “translating” it into terms relevant to the project. The *Gatekeeper* may also “leak” project information to relevant outsiders. This role can also manage the development interface to marketing and to the corporate control structure.

Resulting Context:

This pattern provides balance for the pattern *Firewalls* on page 20, and complements the pattern *Engage Customer* on page 17 (to the degree *Customers* are still viewed as outsiders). *Gatekeeper* and *Firewall* alone are insufficient to protect developers in an organization whose culture allows marketing to drive development schedules.

Design Rationale:

Gatekeeper is a pattern that facilitates effective flow of useful information; *Firewalls* restricts flow of detracting information.

The value of the *Gatekeeper* pattern has been verified in practice. In the discussion of this pattern at PLoP/94, many of the reviewers noted that creating a *Gatekeeper* role had served them well.

Engineers are lousy communicators as a lot; it's important to leverage the communication abilities of an effective communicating engineer when one is found.

Alexander notes that while it is important to build subcultures in a society (as we are building a subculture here in the framework of a company, or of the software industry as a whole), such a subculture should not be closed ("Mosaic of Subcultures", §8); also, cp. Alexander's pattern "Main Gateways" (§53). One might muse that the Gatekeeper takes an outsider through any rites of passage necessary for more intimate access to the development team, by analogy to Alexander's "Entrance Transition" (§112).

Gatekeeper can serve the role of "pedagogue" as in Alexander's pattern "Network of Learning" (§18).

24. Pattern Name: Shaping Circulation Realms

Problem: Patterns of interaction in an organization are not as they should be, as prescribed by other patterns.

Context:

This pattern is a means to implementing ensuing patterns in the language, including *Organization Follows Market* on page 10, *Developer Controls Process* on page 11, *Architect Controls Product* on page 13, *Engage QA* on page 16, *Engage Customer* on page 17, *Buffalo Mountain* on page 23, and others. This pattern may also apply to circulation realms outside the project, such as *Firewalls* on page 20, and many others.

Forces:

Proper communication structures between roles are key to organizational success.

Communication can't be controlled from a single role; at least two roles must be involved.

Communication patterns can't be dictated; some second-order force must be present to encourage them.

Communication follows semantic coupling between responsibilities.

Solution:

Give people titles that creates a hierarchy or pecking order whose structure reflects the desired taxonomy.

Give people job responsibilities that suggest the appropriate interactions between roles (see also *Move Responsibilities* on page 21).

Physically collocate people whom you wish to have close communication coupling (this is the dual of the pattern *Organization Follows Location* on page 9).

Tell people what to do and with whom they should interact; people will usually try to respect your wishes if you ask them to do something reasonable that is within their purview and power.

Resulting Context:

The goal is to produce an organization with higher overall cohesion, with sub-parts that are as internally cohesive and externally de-coupled as possible.

Design Rationale:

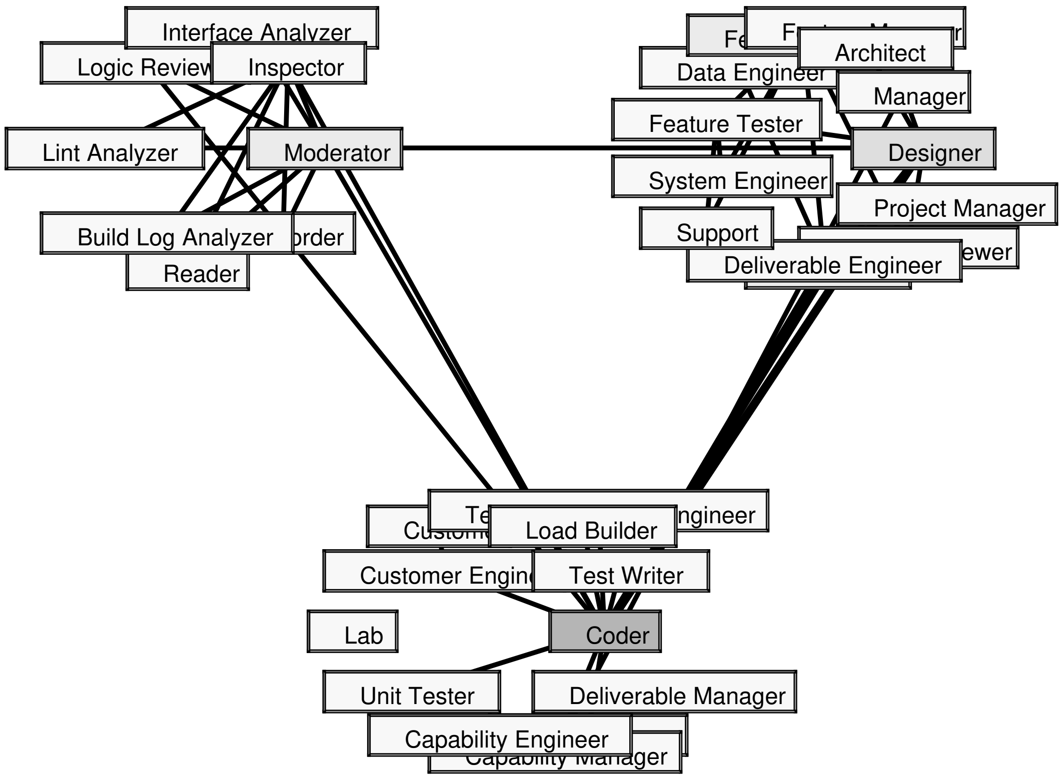
This follows an Alexandrine pattern (§98) of the same name, and has strong analogies to the rationales of "House Cluster" (§37). The same rationale can be found in Thomas Allen. Note that *Move Responsibilities* on page 21 is a closely related pattern.

25. Pattern Name: Move Responsibilities

Problem: Unscrutinized relationships between roles can lead to undesirable coupling at the organizational level.

Context:

Any organization or process.



Coupling in this partitioned design/coding process can be reduced by rearranging Coder's responsibilities (see Pattern 25)

Forces:

You want cohesive roles.

You want cohesive organizations.

De-coupled organizations are more important than cohesive roles.

There may be fundamental trade-offs between coupling and cohesion.

Moving an entire role from one process or organization to another doesn't reduce the overall coupling, but only moves the source.

Solution:

Move responsibilities from the role that creates the most undesirable coupling, to the roles coupled to it from other processes. Simply said, this is load balancing. The responsibilities should not be shifted arbitrarily; a chief programmer team organization is one good way to implement this pattern (in the context for *Developer* role responsibilities).

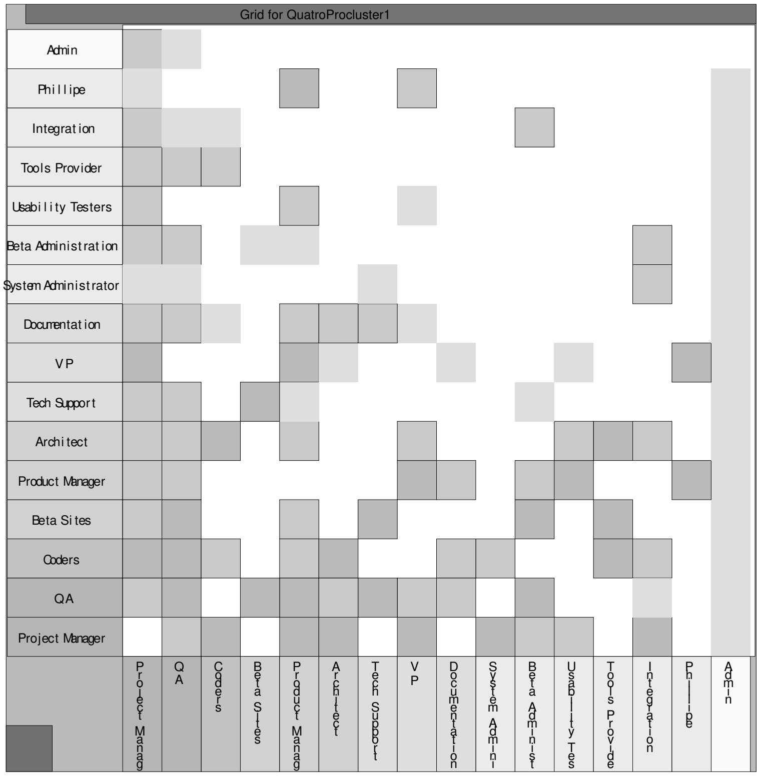
Resulting Context:

The new process may exhibit more highly de-coupled groups. It is important to balance group cohesion with the de-coupling, and this pattern must be applied with extreme care. For example, the *Developer* role is often the locus for a large fraction of project responsibilities, so the role appears overloaded. Arbitrarily shifting *Developer* responsibilities to other roles can introduce communication overhead. A chief programmer team approach to the solution helps balance these forces.

Buffalo Mountain on page 23 is an alternative load-balancing pattern.

Design Rationale:

Most of the design rationale follows from the forces themselves. This is isomorphic to Mackenzie's model that task interdependencies, together with the interdependencies of task resources and their characteristics, define project roles.[22]



Buffalo Mountain (see Pattern 26)

26. Pattern Name: Buffalo Mountain[23]

Problem: We want to optimize communications in a large software development organization, whose members are working together on a common product.

Context:

A development organization straddling several domains, where effective interpersonal communication is key to project success.

A nominal "hub" may already have been established for the organization.

Forces:

Communication overhead goes up non-linearly with the number of people.

Information starvation or role isolation cause people to develop unsuitable sub-products.

Being a communication bottleneck leaves no time to do work.

Communication bottlenecks cause queues in organizational workflows that keep other people from doing work.

Fully distributed control tends to lead to control breakdown.

---

22. Mackenzie, K. D. "Organizing High Technology Operations for Success." *Managing High Technology Decisions for Success*, J. R. Callahan and G. H Haines, eds., 1986.

23. The name from a similarity between the visual graph, and the characteristic shape of a mountain in Colorado, and from the analogies that can be made between the forces contributing to each.

Solution:

1. For any significant project interaction, the distance of the two collaborating roles from the “center” of the organization should sum to a distance less than the smallest distance necessary to span the entire organization.
2. Avoid coupling with neighbors (those equidistant from the center of the process as yourself; i.e., those equally coupled to the process as a whole as yourself) if you are in the outlying 50% of the organization.
3. The intensity of any interaction should be inversely proportional to the sum of the differences of the roles' distance to the center of the process.

Means A: Shuffle responsibilities between roles in a way that moves the associated collaborations, to effect the above patterns (pattern *Move Responsibilities* on page 21). This is Robert Lai's idea; it's simple, but profoundly effective.

Means B: Physically relocate people to enhance their opportunity to communicate (see *Shaping Circulation Realms* on page 21).

Means C: Increase span of control of a role in the project (akin to merging multiple roles into one). It is probably best to Merge Roles with Similar Responsibilities, or better, to Merge Roles with Similar Collaborations. Hey, those are patterns too!

Resulting Context:

The new organization has more balanced communication across its roles.

Design Rationale:

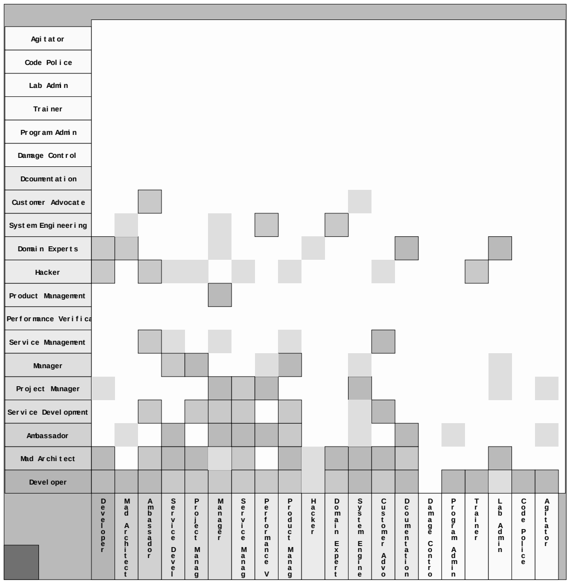
Most of these patterns are empirical, rather than being derived from first principles. I think it important to recognize this path to patterns as a positive and potentially fruitful one. If it works, go with it.

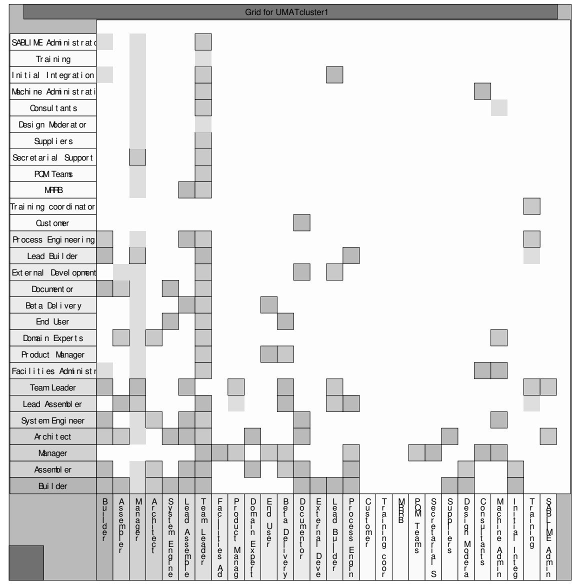
Sub-pattern 1 is empirical. It is the dominant sub-pattern. It infuses a level of “distributed control with central tendency” that lends overall direction and cohesion to an organization. Another way of stating sub-pattern 1 is that most points on the interaction grid tend to live near the axes.

Sub-pattern 2 avoids cliquish splinter groups. It also helps avoid linear event ordering in the distant (support) parts of a process. Linear event ordering (or pipelining) causes points on the interaction grid to line up right below the diagonal. A pattern that avoids points on the diagonal is likely to encourage more parallelism and independence.

Sub-pattern 3 tempers sub-pattern 1, allowing points further from the diagonal but not too far from the origin. This allows for a tight “core” at the middle of the process. It also helps to even the distribution of load across roles in the process. Many organizations are bimodal: they interact tightly in the core, and virtually not at all in the outlying roles. This evens the load across all roles.

The overall (and difficult-to-explain) nature of these sub-patterns is that they improve product quality and reduce time-to-market. They tend to correlate with high spans of control. That in turn reduces the number of people necessary to complete a project, further reducing communication overhead, improving cohesion, and causing the pattern to recursively feed on itself. It's wondrous to watch this happen in an organization.





The organization at the left has a healthy distribution of inward-directed inputs; on the right, the center is overloaded, and turns work requests outward. On the left, core roles <u>do</u> work; on the right, core roles <u>make</u> work (see Pattern 27).

<u>27. Pattern Name: Work flows inward</u>

Problem: Work that adds value directly to the product should be done by authoritarian roles.

Context:

An existing organization where the flow of information can be analyzed. The organization exhibits a management pecking order.

Forces:

Some centralized control and direction are necessary

During software production, the work bottleneck of a system should be at its center (in the adjacency diagram sense). If the center of the communication center of the organization generates work more than it does work, then organization performance can become unpredictable and sporadic.

The developer is already sensitized to market needs through *Firewalls* on page 20, and *Firewalls* on page 20 (no centralized role need fill this function).

Solution:

Work should be generated by customers, filter through supporting roles, and be carried out by implementation experts at the center. You should not put managers at the center of the communication grid: they will become overloaded and make decisions that are less well-considered, and they will make decisions that don't take day-to-day dynamics into account.

Resulting Context:

An organization whose communication grid has more points below the diagonal than above it (see the figures above).

Design Rationale:

Katz & Kahn's analysis of organizations shows that the exercise of control is not a zero-sum game.[24]

The work should focus at the center of the process; the center of the process should focus on value-added activities (*Developer Controls Process* on page 11).

Organizations run by professional managers tend to have repeatable business processes, but don't seem to reach the same productivity plateaus of organizations run by engineers. In programmer-centric organizations, the value-added roles are at the center of the process (*Developer Controls Process* on page 11; *Architect Controls Product* on page 13). The manager should facilitate and support (*Patron* on page 12; *Firewalls* on page 20).

Mackenzie characterizes this pattern using *M-curves*, that model the percentage of task processes of each task process law level (planning, directing, and execution) as a function of the classification.[25]

The rationale is supported with empirical observations from existing projects.

28. Pattern Name: 3 to 7 Helpers per Role

Problem: Uneven distribution of communication.

Context:

An organization whose basic social network has been built.

Forces:

Too much coupling to any given role, and it is in overload.

Too little coupling, and the role can become information-starved and under-utilized.

Solution:

Ensure that each role has between 3 and 7 helpers.

Resulting Context:

A more balanced organization, with better load-sharing and fewer isolated roles.

Design Rationale:

Our empirical results from the organizations studied in the Pasteur project show that any given role can sustain at most 7 long-term relationships. In particularly productive organizations, the number can be as high as 9. Particular needs might suggest that the process designer go outside these bounds, if doing so is supported with a suitable rationale.

29. Pattern Name: Named Stable Bases

Problem: How frequently do you integrate?

Context:

A schedule framework has been determined.

Forces:

Continuous integration, and developers try to follow a moving target.

Too long between integrations, and developers become blocked from making progress beyond the limits of the last base.

Stability is a good thing.

Progress should be made.

Progress must be perceived.

Solution:

Stabilize system interfaces–the architecture–no more frequently than once a week.

---

24. Katz and Kahn, op. cit., p. 314.
25. Mackenzie, *op. cit.*

Resulting Context:

The project has targets to shoot for. This affects the *Customer* view of the process, and has strong ramifications for the *Architect* as well.

Design Rationale:

See the description of the forces. The pattern owes to Dennis DeBruler at AT&T.

It can be helpful to have, simultaneously, various bases at different levels of stability. For example, one AT&T project had a nightly build (which is guaranteed only to have compiled), a weekly integration test build (which is guaranteed to have passed system-wide sanity tests), and a (roughly biweekly) service test build (that is considered stable enough for *QA*'s system test).

30. Pattern Name: Divide and Conquer

Problem: Organizations grow to the point where they cannot easily manage themselves.

Context:

The roles have been defined for a process and organization, and the interactions between them are understood.

Forces:

If an organization is too large, it can't be managed.
Incohesive organizations are confusing and engender dilution of focus.
Separation of concerns is good.
It is useful to have organization boundaries that are somehow lightweight.

Solution:

Find clusters of roles that have strong mutual coupling, but that are loosely coupled to the rest of the organization. Form a separate organization and process from those roles.

Resulting Context:

Each new suborganization is a largely independent entity to which the remaining patterns in this language can be independently applied.

Design Rationale:

See the description of the forces. Note that each sub-organizations that arises from this pattern is fodder for most other patterns, since each subsystem is a system in itself. Also, to see an organization that has been reverse engineered and redivided into new processes, see the picture for the pattern *Move Responsibilities* on page 21.

31. Pattern Name: De-couple Stages

Problem: How do you de-couple stages (architecture, design, coding) in a development process?

Context:

A design and implementation process for a well-understood domain.

Forces:

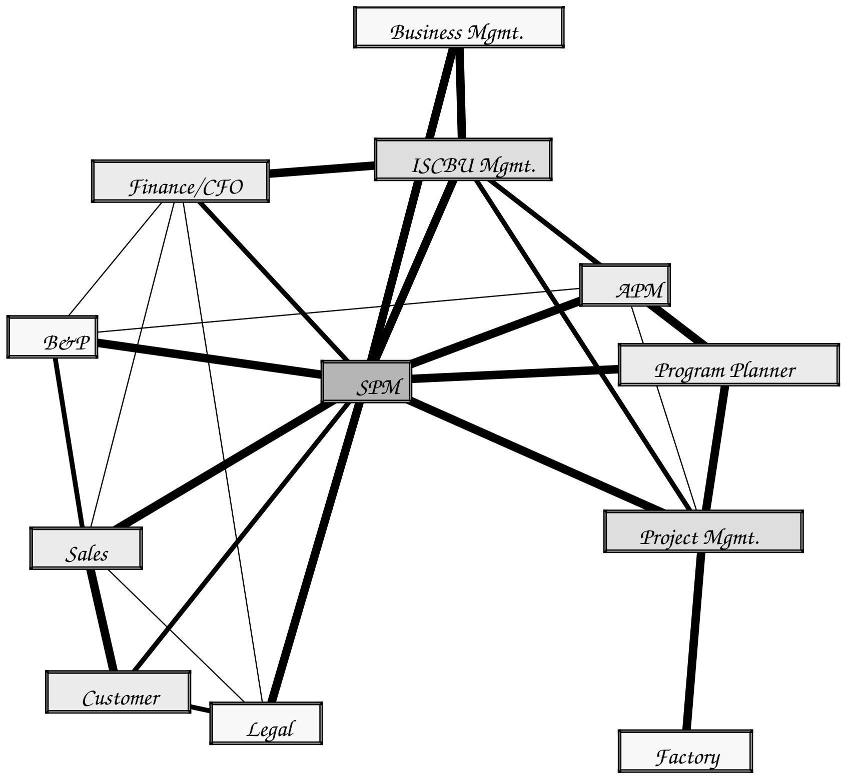
Stages should be independent to reduce coupling and promote independence between stages.
Independence hampers information flow.
Independence creates opportunities for parallelism.

Solution:

For known and mature domains, serialize the steps. Handoffs between steps should take place via well-defined interfaces. This makes it possible to automate one or more of the steps, or to create a pattern that lets inexpert staff carry out the step.

Resulting Context:

The new organization allows for specialization in carrying out parts of the process, rather than emphasizing specialization in solving the customer problem. This approach is "safe" only for well-understood domains, where the mapping from needs to implementation is straightforward. Domains that are well-understood are also



A single central role orchestrates this front-end process that supports sales and marketing activities in this highly responsive process (see Pattern 32).

---

good candidates for mechanization. For less mature domains, the process should build on the creativity of those involved at each stage of the process, and there should be more parallelism and interworking.

This pattern prepares for *Hub-Spoke-and-Rim* on page 28.

Design Rationale:

See the forces.

32. Pattern Name: Hub-Spoke-and-Rim

Problem: How do you de-couple stages (architecture, design, coding) in a serialized development process while maintaining responsiveness?

Context:

A design and implementation process. A well-understood domain. *De-couple Stages* on page 27 has been applied.

Forces:

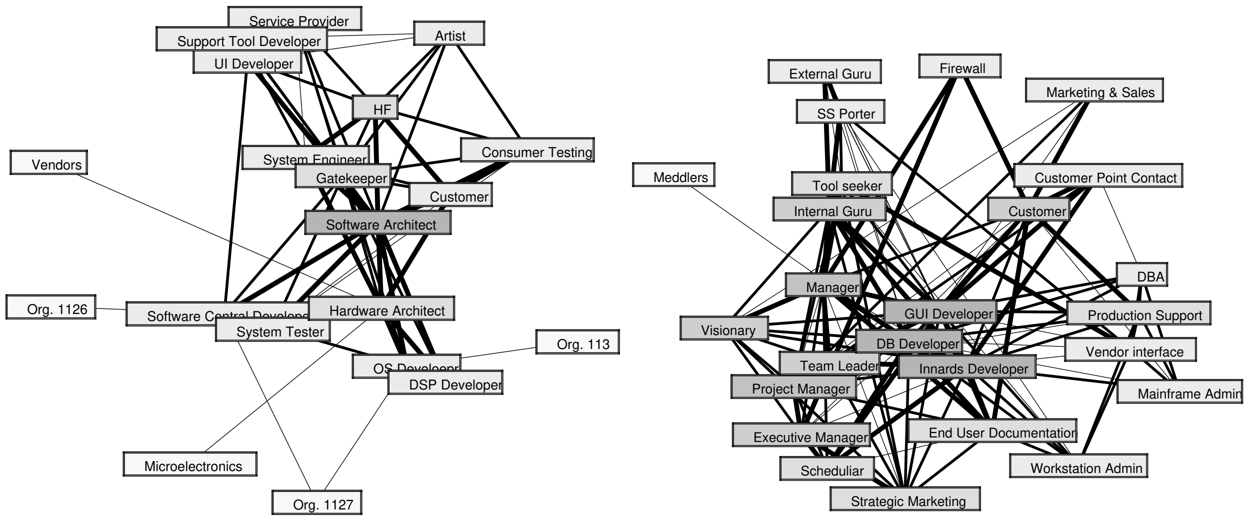
Stages should be independent to reduce coupling and promote independence between stages.

Independence hampers information flow.

Independence creates opportunities for parallelism.

Solution:

Link each role to a central role that orchestrates process activities. Parallelism can be re-introduced if the central role pipelines activities.



The organization on the left has no apparent structure, and though it is productive, it is not likely to evolve well. The organization on the right has no well-partitioned structures, but one can identify logical partitions within it (customer, developer, management, etc.; see Pattern 33.)

Resulting Context:

The process has more order and is more likely to be repeatable than *De-couple Stages* on page 27 alone. The process designer must be wary of the central role becoming a bottleneck, and address such bottlenecks with other patterns (e.g., *Move Responsibilities* on page 21).

Design Rationale:

Empirical studies done on a front-end process called CNM for a large AT&T project (unpublished work); pipelining theory.

33. Pattern Name: Aesthetic Pattern

Problem: An organization has an irregular structure

Context:

The organization has been designed through preceding patterns. The project is in early development of the first release of its product. The organization must plan how to evolve itself beyond the product's first release, and eventually into the product's maintenance phase.

Forces:

Even distribution of responsibility is good because it distributes the work load.

Regular structures, such as hierarchies, can easily be grown by adding more people, without destroying the spirit of the original structure.

A regular hierarchical structure does not distribute responsibility evenly.

Solution:

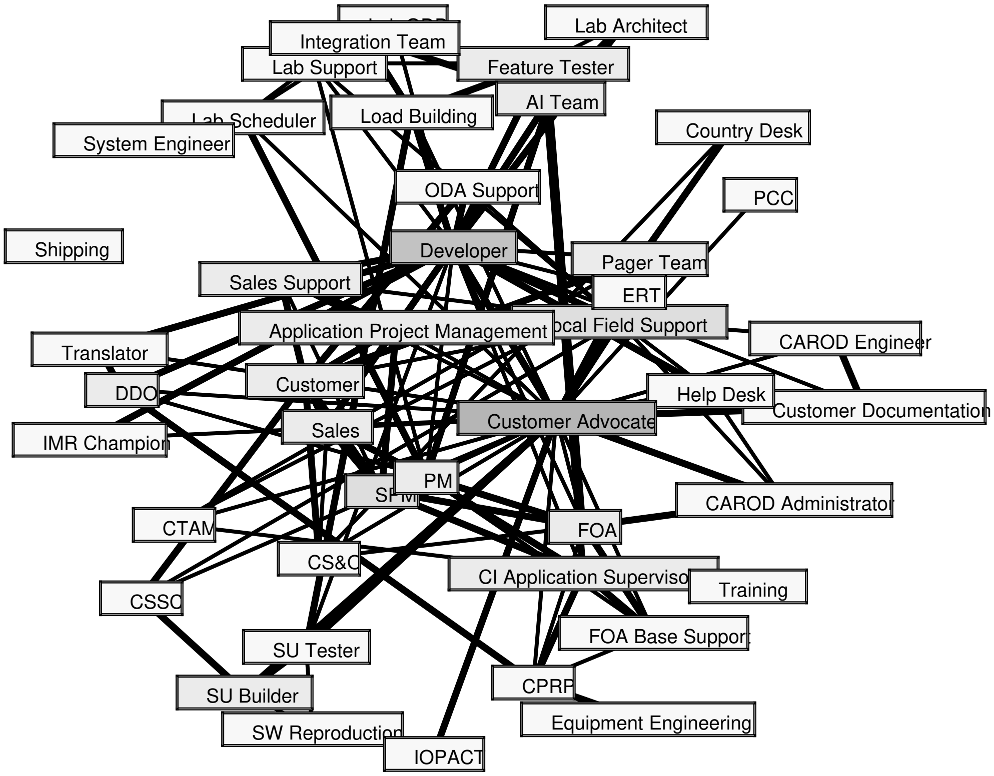
Make sure the organization has identifiable sub-domains that can grow into departments of their own as the project thrives and expands to serve a maintained market.

Resulting Context:

The organization will have sub-organization foundations on which to grow.

Design Rationale:

This is empirical from our organization studies. To this point, patterns help ensure a functional, highly productive organization. But work still must be done to allow the organization to grow elegantly. We achieve that end by



This is a highly responsive process, which owes largely to its high degree of internal coupling (see Pattern 34)

---

identifying the roots of sub-organizations in the current organization. If we can find none, the organization may not be able to grow. For example, it is difficult to grow a Chief Programmer Team organization.

34. Pattern Name: Coupling Decreases Latency

Problem: The process is not responsive enough; development intervals are too long; market windows are not met.

Context:

A service process and, perhaps in special cases, a small design/implementation process using an iterative or incremental approach.

Forces:

Stages should be independent to reduce coupling and promote independence.
Independence improves opportunities for parallelism.
Independence hampers information flow.

Solution:

Open communication paths between roles to increase the overall coupling/role ratio, particularly between central process roles.

Resulting Context:

Coupling of course increases dependence between roles, which may not always be a good thing.
This pattern is somehow related to *Interrupts Unjam Blocking* on page 32.

Design Rationale:

Basic software engineering principle.

35. Pattern Name: Prototype

Problem: Early acquired requirements are difficult to validate without testing.

Context:

Trying to gather requirements necessary for test planning, as in pattern *Application Design is Bounded By Test Design* on page 15, and for the architecture, as for the pattern *Architect Controls Product* on page 13.

Forces:

Requirements are always changing.
Written requirements are usually too ambiguous.
Need to get requirements changes early.
Designers and implementors must understand requirements directly.

Solution:

Build a prototype, whose purpose is to understand requirements. Prototypes are particularly useful for external interfaces. Throw the prototype away when you’re done.

Resulting Context:

A better assessment of requirements to supplement use cases. This pattern nicely complements *Engage Customer* on page 17, and *Scenarios Define Problem* on page 18.

Design Rationale:

The processes of the visualizations illustrating the pattern *Developer Controls Process* on page 11, and the pattern *Engage QA* on page 16, are based largely on prototyping.

“The best friend of the architect is the pencil in the drafting room, and the sledgehammer on the job.” – Frank Lloyd Wright, quoted in *Building with Frank Lloyd Wright*, by Herbert Jacobs, Chronicle Books, 1978.

36. Pattern Name: Take No Small Slips

Problem: How long should the project take?

Context:

The product is under way and progress must be tracked.

Forces:

Too long, and developers become complacent, and you miss market windows.
Too short, and developers become burned out, and you miss market windows.
Projects without schedule motivation tend to go on forever, or spend too much time polishing details that are either irrelevant or don’t serve customer needs.

Solution:

“We found a good way to live by ‘Take no small slips’ from... ‘The Mythical Man Month.’ Every week, measure how close the critical path (at least) of the schedule is doing. If it’s three days beyond schedule, track a ‘delusion index’ of three days. When the delusion index gets too ludicrous, then slip the schedule. This helps avoid churning the schedule.” – Chisholm.

Resulting Context:

A project with a flexible target date. Dates are always difficult to estimate; DeMarco notes that one of the most serious signs of a problem in trouble is a schedule worked backward from an end date.[26]

Design Rationale:

MIT project management simulation; QPW; Brooks.[27]

---

26. DeMarco, Tom. Talk at CaseWorld, Boston, Mass., January, 1993.
27. Brooks, F. P. *The Mythical Man Month*.

37. Pattern Name: Interrupts Unjam Blocking

Problem: The events and tasks in a process are too complex to schedule development activities as a time-linear sequence.

Context:

A high productivity design/implementation process or low-latency service process.

Forces:

Complete scheduling insight is impossible.

The programmers with the longest development schedules will benefit if more of others' code is done before they try integrating or testing later code, and their interval can't otherwise be shortened (see the pattern *Code Ownership* on page 15).

Solution:

If a role is about to block on a critical resource, interrupt the role that provides that resource so they stop what they're doing to keep you unblocked.

If the overhead is small enough, it doesn't affect throughput. It will always improve local latency.

Resulting Context:

The process should have a higher throughput, again, at the expense of higher coupling. Coupling may have already been facilitated by earlier patterns, such as *Shaping Circulation Realms* on page 21, *Move Responsibilities* on page 21, *Buffalo Mountain* on page 23, and *Coupling Decreases Latency* on page 30.

Design Rationale:

See the forces. Also, empirical from a high productivity process in AT&T. There are strong software engineering (operating system) principles as well.

38. Pattern Name: Don't Interrupt an Interrupt

Problem: The pattern *Interrupts Unjam Blocking* on page 32 is causing people to thrash.

Context:

Execution of pattern for *Interrupts Unjam Blocking* on page 32.

Forces:

One worker will inevitably be blocked on you–you can't do both things at once.

Complete, omniscient foresight and scheduling are unreasonable.

Solution:

If a role is about to block on a critical resource, interrupt the role that provides that resource so they stop what they're doing to keep you unblocked.

If the overhead is small enough, it doesn't affect throughput. It will always improve local latency.

Resulting Context:

Largely unchanged from that emanating from *Interrupts Unjam Blocking* on page 32.

Design Rationale:

This is a simple, though somewhat arbitrary, rule to keep scheduling from becoming an elaborate ceremony.

39. Pattern Name: Compensate Success

Problem: Providing appropriate motivation for success.

Context:

A group of developers meeting tight schedules in a high-payoff market.

Forces:

Schedule motivations tend to be self-fulfilling: a wide range of schedules may be perceived as equally applicable for a given task.

Schedules are poor motivators.

Altruism and egoless teams are quaint, Victorian notions.

Companies often embark on make-or-break projects, and such projects should be managed differently from others.

Disparate rewards motivate those who receive them, but may frustrate their peers.

Solution:

Establish lavish rewards for individuals contributing to make-or-break projects. The entire team (social unit) should receive comparable rewards, to avoid de-motivating individuals who might assess their value by their salary relative to their peers.

A celebration is a particularly effective reward.[28]

Resulting Context:

An organization that focuses less on schedule and more on customer satisfaction and systemic success. Such high rewards may cause individuals to over-extend themselves, leading to personal stress with potential risk to the project.

Design Rationale:

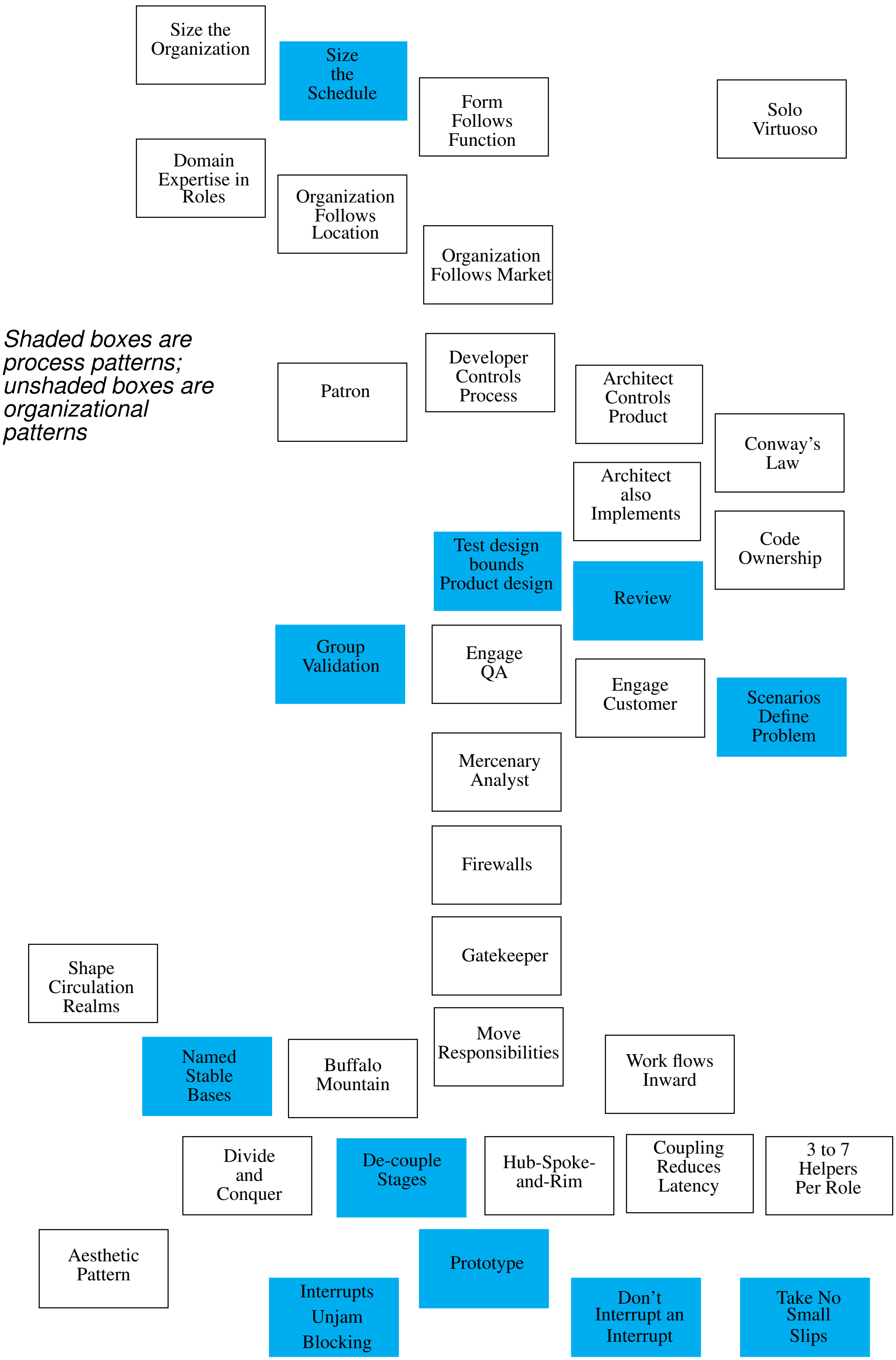
Empirical. There is a strong correlation between wildly successful software projects, and a very lucrative reward structure. Cases include QPW, cases cited at the Risk Derivatives Conference in New York on 6 May 1994; see *Pay and Organization Development*, by Edward E. Lawler, Addison-Wesley, ©1981. The place of reward mechanisms is well-established in the literature.[29]

High rewards to some individuals may still de-motivate their peers, but rewarding on a team basis helps remove the “personal” aspect of this problem, and helps establish the mechanism as a motivator, in addition to being just a post-mortem soother.

DeBruler noted at the PLoP review of this pattern, that most contemporary organization culture derives from the industrial complex of the 1800s, which was patterned after the only working model available at the time: military management. He notes that most American reward mechanisms are geared more toward weeding out problems than toward encouraging solutions. A good working model is that of groups of doctors and lawyers, where managers are paid less than the employees.

---

28. Zuckerman and Hatala, op. cit.
29. Kilmann, R. H. *Beyond the Quick Fix*. San Francisco: Jossey-Bass, 1984.


Size the Organization
Size the Schedule
Form Follows Function
Solo Virtuoso
Domain Expertise in Roles
Organization Follows Location
Organization Follows Market
Shaded boxes are process patterns; unshaded boxes are organizational patterns
Patron
Developer Controls Process
Architect Controls Product
Conway's Law
Architect also Implements
Code Ownership
Test design bounds Product design
Review
Group Validation
Engage QA
Engage Customer
Scenarios Define Problem
Mercenary Analyst
Firewalls
Gatekeeper
Shape Circulation Realms
Named Stable Bases
Buffalo Mountain
Move Responsibilities
Work flows Inward
Divide and Conquer
De-couple Stages
Hub-Spoke-and-Rim
Coupling Reduces Latency
3 to 7 Helpers Per Role
Aesthetic Pattern
Prototype
Interrupts Unjam Blocking
Don't Interrupt an Interrupt
Take No Small Slips