PART II

The Pattern Languages

Finally, the patterns themselves! Thank you for patiently reading the introductory material, for it will help you use the patterns.

We have divided the patterns into four interrelated pattern languages:

1. Project Management: the organizational aspects of managing projects.
2. Piecemeal Growth of the Organization: the ways in which an organization grows and develops over time.
3. Organizational Style: the general approach to the way the organization works.
4. People and Code: the ways in which people affect code—and the ways in which the design of code affects people!

Each pattern language presents patterns in a sequence that allows the patterns to build on each other. Sometimes a pattern recurs in multiple pattern languages, but we present the pattern only in the pattern language with which it is most strongly connected, and we include a reference to that appearance in other appearances of the pattern. In practice, you will use all four of these pattern languages together, weaving patterns together to solve problems and to strengthen your organization one pattern at a time.

The first two pattern languages are design pattern languages; the second two are construction pattern languages. Chapters 4 and 5 are dedicated to these two kinds of patterns, respectively. Alexander makes the same distinction in his pattern language, separating the act of design from the engineering considerations of construction.

Design patterns are those that lay the foundation of the entity to be built—buildings and towns for Alexander, and software development organizations for us.

Construction patterns deal with the nuts and bolts of creating the thing. Organizations need to be built just as surely as buildings need to be built.

The appendix SUMMARY PATLETS (Appendix A) presents summaries of all of the patterns in *patlet* form. A patlet is a terse summary of a pattern's problem and solution. You may find this reference useful as you set about putting the patterns into practice.

CHAPTER 4

Organization Design Patterns

The term *design patterns* has unfortunately come to mean the collection of 23 patterns that appears in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Gamma 1995]. Here, we use the term in the same sense Alexander does in his classic text *A Pattern Language* [Alexander 1977]. In Alexander's sense, a design pattern is something you use to understand the geometry of a building and to understand the major relationships between parts. It is a definition that most of us recognize as applying to the word *architecture* in software.

Once you design an organization, the organization comes to life through organizational *construction patterns.* Construction patterns discuss the materials and processes used to put the conceptual design into practice.

The distinction between these two kinds of patterns isn't as clear in organizational design as it is in the design of buildings, and even there the difference isn't formal or clean. We separated the two kinds of patterns based less on their characterization as "design" or "construction" patterns than according to their affinity for each other. The so-called "construction patterns" can be found in the chapter ORGANIZATION CONSTRUCTION PATTERNS (CHAPTER 5).

4.1 Project Management Pattern Language

Project Management is a crucial part of organizational design. Many organizations have a project manager role, but in fact project management is a much broader function—so broad that it covers almost a quarter of the patterns in this book.

The patterns here do concern themselves with all of the things a project manager worries about. We start out with COMMUNITY OF TRUST (4.1.1), followed by SIZE THE SCHEDULE (4.1.2). In today's markets, time to market is everything. In the classic view of project management, which suggests that there are three resources one can trade off against each other—

staff, functionality, and schedule—schedule is most often the strongest invariant. Past years have seen functionality fall from this first-place position as software development enterprises have come to realize the difficulty in both capturing and meeting detailed a priori requirements. Customers have come to the realization that it's better to get *something* that works in a finite amount of time than to spend a seeming eternity getting it right the first time. Instead, we tend to defer correctness to later releases.

The Pattern Language

Figure 4.1 depicts the patterns in the pattern language and the connections between them. The connections are as much a part of the language as the patterns themselves. Each pattern provides a possible context for any patterns that appear below it. The figure depicts the dependencies between the patterns that govern the order in which they are to be applied. You start at the top

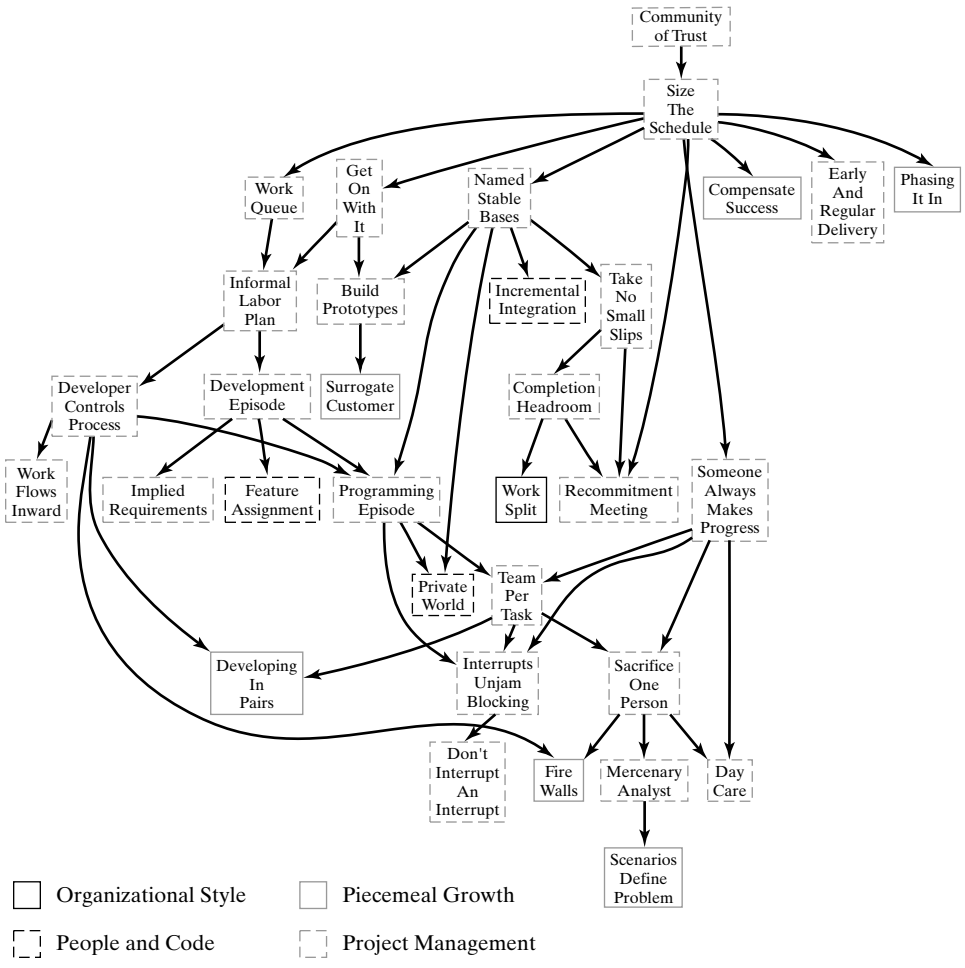


FIGURE 4.1 Patterns and Connections

and work your way toward the bottom. If a pattern has several subtending patterns, you can apply as few or as many of them as you like, and in any order.

The pattern language is based on empirical study of organizations that develop software, most of which deliver some software artifact to a customer. However, the pattern language has little to do with software per se. We believe these patterns reflect management principles that are deeper and broader than principles applied to software alone. Software development organizations can learn from these broader principles.

Here is a real story about a real project that features many of the patterns in this pattern language. Think of this story as a sequence of application of the patterns.

A Story about Project Management

In the mid 1980s, my group embarked on an ambitious project. We took a successful product and adapted it to new technology. We began by testing the concepts in prototypes [BUILD PROTOTYPES (4.1.7)], and their success gave us the confidence to SIZE THE SCHEDULE (4.1.2).

Because we were building on an existing product, it was easy to have NAMED STABLE BASES (4.1.4) of code, and we continued them throughout the project. These stable bases made it possible—and necessary—to provide developers with a way to have their own view of the system, a PRIVATE WORLD (4.1.6). There was ample tool support for these views.

Although the project was large, the project was basically centered on the developers. For example, we decided on our own coding standards [DEVELOPER CONTROLS PROCESS (4.1.17)], which certainly had a feel of WORK FLOWS INWARD (4.1.18). Developers had some latitude about how to organize their work, and WORK QUEUE (4.1.13), INFORMAL LABOR PLAN (4.1.14), and PROGRAMMING EPISODES (4.1.19) were common.

Unfortunately, we had problems. One of the biggest was that we did not allow COMPLETION HEADROOM (4.1.10). As the technical difficulties intensified, the schedule became tighter. Finally, the head of the project called everyone together, announced a single large schedule slip [TAKE NO SMALL SLIPS (4.1.9)], and asked everyone to commit to the new schedule [RECOMMITMENT MEETING (4.1.12)].

We continued to struggle with technical challenges, some of which became crises. We created teams to deal with these crises [TEAM PER TASK (4.1.21)], and even had to SACRIFICE ONE PERSON (4.1.22) on at least one occasion. However, no crisis stopped everyone [SOMEONE ALWAYS MAKES PROGRESS (4.1.20)], in part because the architecture of the system allowed some progress to be made regardless of the struggles we encountered.

In the end, we met the slipped date. But the technology was moving in such a direction that it made no sense to deploy it. However, pieces of that project were used in later projects for years to come.

4.1.1 Community of Trust **

In high school, I went to music camp one year. During one orchestra rehearsal, my section was struggling with a particularly difficult passage. The conductor asked about it, and I said, "Don't worry. We will have it tomorrow." He said, "OK," and continued with the rehearsal. By the next day, we had indeed learned the passage.

... once an organization has been established, interpersonal relationships have a significant positive or negative impact on the effectiveness of the team.

✣ ✣ ✣

It is essential that the people in a team trust each other; otherwise, it will be difficult to get anything done.

Communication is essential to the smooth working of any team. For example, software developers must constantly talk to each other to coordinate interfaces, builds, and tests. If individuals do not trust each other, communication will not be smooth.

If people do not trust each other, they will spend time in defensive mode. For example, if I don't believe you will provide me with a certain interface on time, I might go to great lengths to code around it, thus costing extra work and time.

Design reviews can foment distrust. All too often, design reviews become contests among the reviewers to show who is the most clever and thus do not provide helpful suggestions to the

designer. One alternative is for people to put on their best social behavior in reviews; however, such behavior can dampen the energetic discussions that lead to the best insights in group discussions.

The organization might have policies that seem distrustful. For example, one may have to jump through hoops to be allowed to submit code to the project base.

The perception of trust or mistrust becomes the reality, regardless of the intention.

Therefore:

Do things that explicitly demonstrate trust. Managers, for example, should make it overtly obvious that they facilitate the achievement of organizational goals, rather than playing a central role to assert control over people. Take visible actions to give developers control over the process.

The key here is that the actions must be visible and obvious, particularly if these actions involve the removal of onerous rules and processes. Shortly before I went to work at a certain company, for example, the company dispensed with time clocks for research and development personnel. My co-workers spoke fondly of the time clock smashing ceremony they had.

This pattern is different than the oft-cited "empowerment" strategy. Empowerment is a conscious abdication of control to lower levels [see THE OPEN/CLOSED PRINCIPLE OF TEAMS (6.1.4)]. In a COMMUNITY OF TRUST, progress is more often made by bilateral agreement than by unilateral directions. If people feel they have a voice and have influence over decisions, they are more likely to trust those who make the decisions. By the same token, they are likely to be more responsive in carrying out responsibilities they have committed to themselves than in carrying out responsibilities that have been assigned to them. In fact, you can't give someone responsibility; you can only hold someone accountable. Responsibility is taken, not given. One of the most demoralizing things a manager can do is to give accountability in the absence of resources to responsibly carry out the task.

Trust must be built between the customer and all team members in order to lay out project plans that extend from SIZE THE SCHEDULE (4.1.2) in the PROJECT MANAGEMENT PATTERN LANGUAGE (4.1). The same is true for role differentiation. Encouraging a sense of pride and individuality in every team member can contribute to trust, as in SIZE THE ORGANIZATION (4.2.2) and its subtending patterns in the PIECEMEAL GROWTH PATTERN LANGUAGE (4.2). Build trust by starting small with FEW ROLES (5.1.2), and let this principle guide the ORGANIZATIONAL STYLE PATTERN LANGUAGE (5.1). To keep people from working defensively, one needs a team spirit, which is true of ARCHITECT CONTROLS PRODUCT (5.2.3) and subtending patterns relating to the PEOPLE AND CODE PATTERN LANGUAGE (5.2).

COMMUNITY OF TRUST provides a foundation for many other patterns, such as UNITY OF PURPOSE (4.2.12), PATRON ROLE (4.2.15), FIREWALLS (4.2.9), DEVELOPER CONTROLS PROCESS (4.1.17), RESPONSIBILITIES ENGAGE (5.1.14), and more.

So why is COMMUNITY OF TRUST a separate pattern? It has a specific structural impact: it is about nurturing communication paths, and it also has some positional impact (in particular, it encourages manager roles to shift away from the center). Second, the visible nature of the actions is important, and we haven't captured this visibility in any of the other patterns.

Trust is contagious, and it spreads most effectively through an organization from the top down.

4.1.2 Size the Schedule **

Software engineers determining the next schedule.

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

Both overly ambitious schedules and overly generous schedules have their pains, either for the developers or the customers.

If you make a schedule too generous, developers become complacent, and you miss market windows. But if a schedule is too ambitious, developers become burned out, and you also miss market windows. And if the schedule is too ambitious, product quality suffers, and compromised architectural principles establish a poor foundation for future maintenance.

Conventional wisdom says that you can trade off staff, schedule, and functionality. While principles such as Brooks' "adding people to a late project makes it later" [Brooks 1995] cast doubt on the place of staff in this equation, it's clear that schedule and functionality trade off against each other. Ward Cunningham says in his pattern COMPARABLE WORK, "Every project must commit to delivery on a few hard and fast dates. This is actually fortunate because it is about the only way to get out of work that is going poorly" [Cunningham 1996]. In a reasonable business climate, it is much smarter to hold the schedule constant and to negotiate functionality

than it is to extend the schedule. The customer believes you can cut functionality, but a promise of having the yet unattained functionality at some future date leaves the customer much less comfortable. And projects without schedule motivation tend to go on forever or to spend too much time polishing details that are either irrelevant or that don't serve customer needs.

Therefore:

Reward developers for negotiating a schedule they prove they can meet with financial bonuses [or at-risk compensation; see COMPENSATE SUCCESS (4.2.25)] 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, whereas the internal schedule is negotiated with development staff. The internal schedule should be shorter than the external schedule by 2 or 3 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 renegotiated [RECOMMITMENT MEETING (4.1.12)].

Help delineate the schedule with NAMED STABLE BASES (4.1.4). Grow the schedule as needed with PHASING IT IN (4.2.3). Define initial targets with WORK QUEUE (4.1.13). Make sure SOMEONE ALWAYS MAKES PROGRESS (4.1.20).

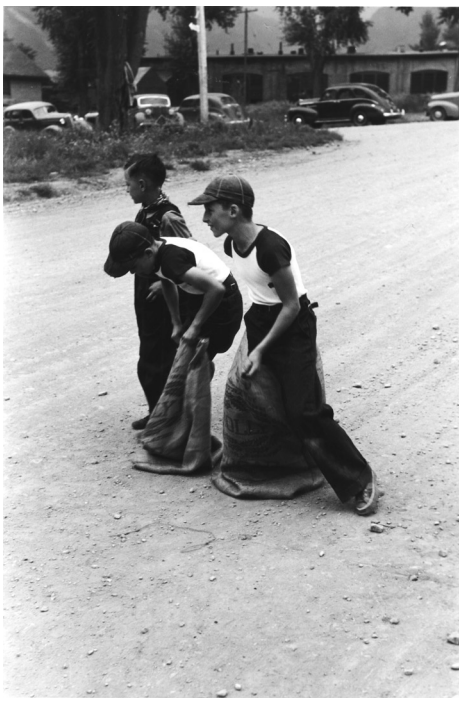
The forces come from the Massachusetts Institute of Technology (MIT) project management simulation and from studies of projects such as Borland Quattro Pro for Windows. Another manager suggested that the skew between the internal and external schedules be closer to 2 months than 2 weeks because slippage usually reflects a major oversight that costs 2 or 3 months.

De Marco talks about rewarding people for accuracy of schedules (see [DeMarco Boehm 1986]). Also, read about the place of promptness in [Zuckerman Hatala 1992].

You don't need a full schedule—or perhaps any schedule at all—to get started. See GET ON WITH IT (4.1.3) and BUILD PROTOTYPES (4.1.7).

4.1.3 Get On with It **

Alias: PARTIAL EVALUATION



Get ready ...

Go!!!!!

During one study, I asked the organization to describe how they develop software. "Well," they said, "project management gives us a list of features they want estimates for. So we start working on the features we think are the most important. Over time, they ask for more detailed estimates, and the features we are working on have smaller estimates because they are underway. Those features generally make the cut. By the time we get official approval to begin development, we are nearly finished."

... you have a good idea of a market need and, furthermore, a good idea of how to get started on parts of the project. You're eager to get started, but you want to proceed deliberately and by the path that will be both expedient and productive.

You can't wait until you have every last requirement to get started.

Team members are sitting idle because their upstream tasks have not been completed. On the one hand, you want requirements to be developed carefully. On the other hand, you have some information, and some people are sitting idle.

Therefore:

As soon as you have some confidence about project direction, start developing areas in which you have high confidence. These areas may involve hardware development (or procurement), algorithm development, database schema development, etc. Let each subgroup work according to an INFORMAL LABOR PLAN (4.1.14) as if the group were in full-swing development.

Note that "high confidence" refers to project direction and requirements, not to technology. It's perfectly all right, and in fact desirable, to work on the technologically risky areas first [see BUILD PROTOTYPES (4.1.7)].

Give yourself some room to retrench later as requirements become more clear.

In many projects, behavioral requirements are one of the last things that designers get right. Many projects ship a first release that meets only basic requirements, with economically more significant requirements being met in subsequent releases. Telecommunications systems often follow this pattern, offering basic communications systems in early releases and more advanced features later. In fact, the impact of behavioral requirements on the overall structure of the system is often overrated. The code that meets behavioral requirements often lives in application code that is added very late to a robust stable base. This base thus reflects deep domain knowledge more than it reflects behavioral requirements. Much common code can be developed early on with high confidence, code that supports common domain functionality that is part of most systems for a given market. This code can often be started or acquired before requirements are firm.

This pattern can increase rework, but it is more in the spirit of piecemeal growth architecture than is a master-planned system that precipitates from complete requirements. It is likely

that any false starts will also be educational at the enterprise level. In fact, as a risk-management measure one can consciously decide not to commit to the results of such an activity. On the enterprise level, this pattern becomes the pattern SKUNKWORKS (4.2.14); at the project level, it is BUILD PROTOTYPES (4.1.7).

There are two occasions in which you cannot tolerate that rework. First, if the task is the process bottleneck, it must work at peak efficiency, and rework should be minimized. Very infrequently, the rework will take longer than the original task, and in such cases this pattern should not be used.

Teams need good communication with their upstream colleagues through the use of patterns like RESPONSIBILITIES ENGAGE (5.1.14) and HALLWAY CHATTER (5.1.15).

A process that does not constrain the overall system can afford to be done inefficiently and in parallel with other processes. It is often the case that the analysts, designers, and programmers can get started right away even if they lack finalized requirements. Serializing their work will take longer than doing 10-20 percent rework. One database group we studied constrained the process: They could not afford rework and had to work in the most efficient way possible. Therefore, they did not start official development early, but instead waited until their requirements were stable. The designers/programmers had enough extra time that they could afford to prototype some test databases for themselves, which were thrown away when the database designers did their final design.

See [Goldratt Cox 1986].

EXAMPLE:

Each team had one requirement and analysis person and two to three designers/programmers. Database design was understaffed and constraining the process, so it was made into a special service group and given final requirements only (the counterforce). A first cut at the requirements had been done earlier, so a rough set of requirements was available. The system was pretty much the same throughout.

The designers/programmers quickly got ahead of the requirements people, who were busy in meetings trying to nail down details of the requirements. If they had waited until the requirements were solid, they would not have enough time to do their work. They were able to guess quite closely what the requirements would be, even without knowing final details, so they started design and programming right away. The requirements people gave them course corrections after each meeting. The amount of time it took to incorporate those mid-course alterations was small compared to the total design time.

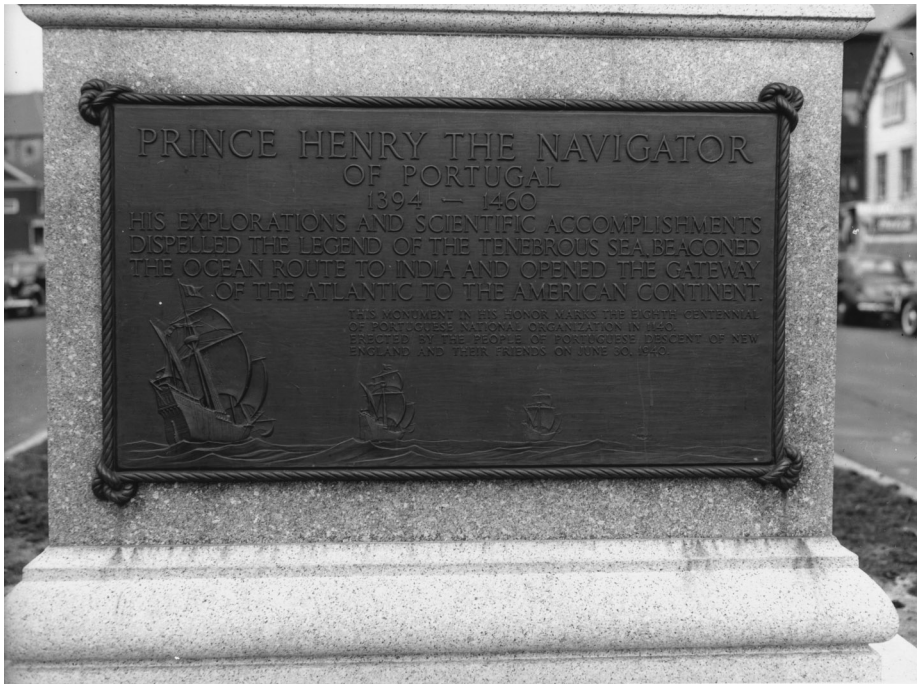
This pattern comes from Alistair Cockburn's original pattern ALL AT ONCE (A.5.3) [Cockburn 1996], which was later modified and renamed GOLD RUSH [Cockburn 1998]. The alias name "PARTIAL EVALUATION" comes from the inspiration that this pattern is a temporal form of DIVIDE AND CONQUER (5.1.6). GET ON WITH IT (4.1.3) arose when we discovered that the name the pattern bore at that time—JUST DO IT—conflicted with another pattern written by Jeff Garland. Garland suggested the current name.

Shalom Reich writes:

> The "ALL AT ONCE" pattern appears to be a typical Project Management "crash project" approach. In a "crash project" one must be careful to identify true predecessors for each task with the goal of reducing the "critical path." This allows parallel efforts to proceed which will all "come together" at the last possible moment. I have found that project plans often contain *false* linkages between

tasks. For example, in one large project we had a "specification" phase. I was able to break the project into several smaller projects which each had its own specification phase. This allowed me to juggle my limited resources and have coders working on the part that went first through the specification phase at the same time that the analysts were working on the specifications for the second sub-project [Reich 2001].

4.1.4 Named Stable Bases *



A stable base with a name on it ...

... the project schedule has been laid out, and development has started.

It is important to integrate software frequently enough so that the base doesn't become stale, but not so frequently that you damage a shared understanding of what functionality is sound and trusted in an evolving software base.

If you try continuous integration, developers struggle to follow a moving target, and there is no shared sense of quanta of functionality at any given time or quanta of progress from week to week. But if it's too long between integrations, developers become blocked from making progress beyond the limits of the last base.

So, while stability is a good thing, the project must always make progress—and, more importantly, the stakeholders must *perceive* that progress is being made.

Therefore:

Stabilize system interfaces—the architecture—about once a week. Give the stable system a name of some kind by which developers can identify their shared understanding of that version's functionality.

The names need not be elaborate; they can, for example, simply be a load number. The names should, however, be easy to remember, easy to identify with the correct version of software, and easy to distinguish from each other. The idea is to provide some sort of handle that people can use to communicate about a stable base.

Other software can be changed (and even integrated) more frequently.

A prototype can be an expedient for one of the NAMED STABLE BASES [see BUILD PROTOTYPES (4.1.7)].

The project has targets to shoot for and benchmarks whose accomplishments can be trumpeted to customers. These targets and benchmarks affect the Customer view of the process and have strong ramifications for the Architect as well.

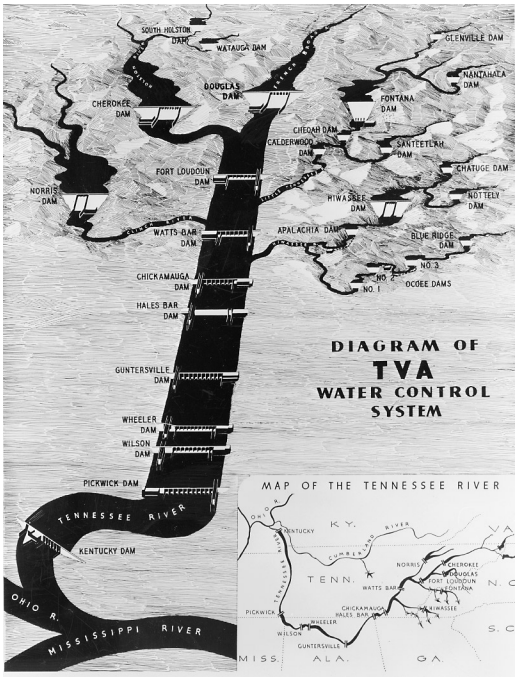
The pattern was initially pointed out by Dennis DeBruler at AT&T.

The main point of the pattern is that a project should schedule change introduction so the effects of changes can be anticipated. It is less important to publish the content of a change (which will go unheeded under high change volume) than to ensure that the development community understands that change is taking place. It is important not to violate "the rule of least surprise."

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 systemwide sanity tests), and a (roughly) biweekly service test build (which is considered stable enough for quality assurance (QA) system testing).

PROGRAMMING EPISODES (4.1.19) is an example of this pattern in the small.

4.1.5 Incremental Integration **



Contribute to software one piece at a time, gradually, avoiding waterfalls and other precipitous changes.

... some organizations have infrequent integrations that reflect large changes, which can make it difficult for the integration release to work as expected, complicate the process of work integration, and make NAMED STABLE BASES (4.1.4) difficult to achieve when modules do not work together. Because we often develop with one OWNER PER DELIVERABLE (A.5.19), there will be occasional mismatches between development units.

For iterative development to work well, it is necessary to make sure that components work together.

Subsystems are developed at different rates. Developers work in a PRIVATE WORLD (4.1.6). We need to find a way to make it possible to integrate without surprises.

Therefore:

Provide a mechanism to allow developers to build all of the current software periodically. Developers should be discouraged from maintaining long intervals between check-ins.

Developers should at any time also be able to build against any of the NAMED STABLE BASES (4.1.4) or the newest checked-in software.

Assign the task of building the entire software system periodically. NAMED STABLE BASES (4.1.4) suggests intervals that are no more frequent than 1 week. This periodic build should be checked for interface compatibility (does it compile?) and tested (does it still work?).

Encourage developers to build from files that are likely to be in the release in order to anticipate and allow time to correct for incompatibilities. The goal is to avoid a "big bang" integration and to allow the developmental build to proceed smoothly.

This pattern can be combined with PRIVATE WORLD (4.1.6) to ensure that the changes integrate with a copy of the current development system. There are issues relating to the size of the software system (some systems take quite a while to build, making frequent integrations difficult). Address these issues with PRIVATE VERSIONING (5.2.16) to allow the developer some leeway on deciding when to integrate new code into the environment, but do not put the issues off for too long.

EXAMPLE:

The developer's work space could be updated (at the developer's request) to a named stable base from the project repository on approximately a weekly basis. The developer will also retrieve the current files from the repository to anticipate how the current changes in the work space will integrate with files that may later be in the baseline.

This pattern was derived from INCREMENTAL INTEGRATION in [Berczuk Appleton 2002].

4.1.6 Private World **

... an organization is creating NAMED STABLE BASES (4.1.4), and developers can build against these versions, integrating their own code with the latest other code [INCREMENTAL INTEGRATION (4.1.5)].

How can we balance the need for developers to use current revisions, based on periodic baselines, with the desire to prevent developers from experiencing undue grief by having development dependencies change from underneath them?

It is important for developers to work with current versions of software subsystems in order to keep up with the latest enhancements, avoid running into bugs that have already been fixed elsewhere, and avoid getting out of synch with interface changes.

Introducing new software into an environment while debugging may cause grief by introducing new behavior and providing distractions. Because of the time spent resolving integration issues in some cases, code may no longer compile due to interface changes.

However, we must balance the need to keep up to date with the need of developers to maintain a stable environment for feature development/bug fixing.

Some organizations facilitate INCREMENTAL INTEGRATION (4.1.5) by having a shared baseline of code, libraries, etc. Unfortunately, changing a code base, even in a different subsystem,

can cause problems when there are interface changes, for example. You want to avoid hearing stories about developers leaving a problem at night in order to view it in the morning with a clear head, only to find that the test environment does not compile in the morning.

Therefore:

Provide a mechanism where developers can maintain a PRIVATE WORLD development environment. In their PRIVATE WORLD, they can control the rate of integration, which allows them to avoid having an integration step interrupt work in progress. The environment should represent a snapshot of all of the software being developed in a system, not just the code the developer is modifying. Try to ensure that the private development area is not used as a means of avoiding integration issues.

A starting point for the independent development area would be one of the NAMED STABLE BASES (4.1.4) that have been previously released. Developers then build their software and any related software that depends on their software. Alternatively, you can provide a capability that allows developers to perform a private system build from source code (and other artifacts).

While allowing developers the freedom to decide when to allow changes into their space, you need to make sure that the developers update their code as often as possible to avoid integration surprises. So, encourage developers to integrate their code frequently, perhaps by providing a mechanism for easily backing out of a difficult change.

Depending on the details of implementation, one consequence of using this pattern might be that project disk space requirements may grow quickly, since developers will have their own copies of the source code. But the costs of personnel always exceed the cost of an extra disk. A modification to this approach is that stable and distantly related subsystems can be used by reference, but one should be made aware when changes are imminent. In this case, the configuration management system should provide access to prior NAMED STABLE BASES (4.1.4) as well.

Developers can simply refrain from advancing to a new instance of the NAMED STABLE BASES until the current problem is solved.

A variation of a PRIVATE WORLD is a shared integration machine. In this case, the developers move their new code to a system that has a current version of the system.

The pattern simulates SOLO VIRTUOSO (4.2.5). See also PRIVATE VERSIONING (5.2.16).

EXAMPLE:

A developer is working on a problem. The developer's work space is self-contained with all of the files needed to build the system. The developer's work space is updated after the problem is solved in the context of the current NAMED STABLE BASES and only at the developer's request.

NOTES:

Brad Appleton points out:

> Sun's NSE (Network Software Environment) had this type of thing built into it. I think that the more recent TeamWare product may also have preserved some of

these concepts. NSE let you create work spaces that it called "environments." There were three kinds of environments you could create:

- Independent Development Environments: for Independent Development.
- Independent Integration Environments: for integrating (importing and merging) and reconciling changes and integration building and testing.
- Independent Release Environments: for release builds, system test, and other release engineering and software product deployment activities [Appleton 2001].

PRIVATE WORLD captures the spirit of all these environments.

An environment would insulate developers, but would not isolate them. There was an event-notification and registration mechanism for broadcasting events in one or more other environments to interested parties (maybe this is a more general configuration management event-notification pattern of which things like baseline publishing and change publishing are concrete variants).

This pattern was derived from PRIVATE WORLD in [Berczuk Appleton 2002].

4.1.7 Build Prototypes **

... you are trying to gather requirements necessary for test planning, as in the pattern Application Design Is Bounded by Test Design (4.2.30), and for the architecture, as in the pattern Architect Also Implements (5.2.10). Some of these requirements come from the customer, but some are design decisions that come from the structure of the solution itself. For example, you may be building a user interface, developing some new database or network technology, working on a new, critical algorithm, or lacking an understanding of your project domain.

A project must test requirements and design decisions in order to reduce the risk of wasted cost and missed expectations.

You need knowledge to proceed on development, and you must move forward; yet, requirements (or your understanding of them) are always changing.

You're missing information about the product (not the process), you have a best guess you can use to move forward, and you want some way to evaluate the result of your best guess.

Written requirements that are gathered once at the beginning of a development cycle with the hope that they can drive development are usually too ambiguous.

You want to get requirements changes as early as possible, and you want an understanding of requirements to lead deployment as much as possible.

Designers and implementors must understand requirements directly—the fact that the requirements have been captured in a document isn't enough. And the ability of designers and

developers to understand requirements implies that they must understand the implementation ramifications.

Therefore:

Build an isolated prototype solution whose purposes are to

- **Understand requirements, including latent needs**
- **Validate requirements with customers, as in ENGAGE CUSTOMERS (4.2.6)**
- **Explore human/computer interactions for the system**
- **Explore the cost and benefits of design decisions**

The prototype is a small system that explores a small number of issues in isolation using best current knowledge. By examining that small system, you can learn whether or not your current knowledge is correct and sufficient. Prototypes are particularly useful for external interfaces.

Throw the prototype away when you're done. This action is more important than it may sound. Since the purpose of prototyping is to gain knowledge, prototypes can (and should) ignore details necessary in production software. Yet, such details (e.g., scale, performance, and robustness) cannot be incorporated into prototype-based software without the result resembling the proverbial bowl of pasta.

You will decide if your current knowledge is sufficient. If it is, adapt that small system's design (not its code!) to your larger system (incorporate it entirely if it was built to production specifications). If not, decide if you now have enough information to safely proceed or if you need to develop another prototype.

It's good to use DEVELOPING IN PAIRS (4.2.28), particularly if one of the pair represents the customer interests or is a customer per se.

Prototypes are a good supplement to use cases to help assess requirements more thoroughly. For one thing, prototypes help bring unstated requirements into the open. This pattern nicely complements ENGAGE CUSTOMERS (4.2.6) and SCENARIOS DEFINE PROBLEM (4.2.8).

The visualizations used for DEVELOPER CONTROLS PROCESS (4.1.17) and the pattern ENGAGE QUALITY ASSURANCE (4.2.29) come from processes based largely on prototyping.

Continued prototyping without convergence means that the design is constantly shifting and that the team is not learning enough to reach a conclusion. If other teams that depend on the prototyping team do not get the stable interface they need, it is time to get out of prototyping and either implement the system or ENGAGE CUSTOMERS (4.2.6) [RECOMMITMENT MEETING (4.1.12)] to evaluate current project directions and priorities.

There are subtle organizational overtones to building prototypes. It is important that the ARCHITECT CONTROLS PRODUCT (5.2.3), instead of the prototype controlling the product. Therefore, the prototyping team should be kept separate from the Architect and the ARCHITECTURE TEAM (5.2.4). Instead, the prototyping activity helps enhance the DOMAIN EXPERTISE IN ROLES (4.2.22). And one of the positive effects of building a prototype is that it reduces the risk of the unknown. The prototype helps to define the scope of the problem, as well as to offer a possible solution.

RELATED PATTERNS:

- EARLY AND REGULAR DELIVERY (A.5.11)—adds knowledge about your development process.
- MICROCOSM (A.5.18)—returns measurable data about process and technology.

Another related pattern is Alistair Cockburn's CLEAR THE FOG (A.5.7) [Cockburn 1998], which one might view as a generic version of this pattern. In that pattern, he recommends, "Do something (almost anything) that is a best initial attempt to deliver some part of the system in a short period of time" in the interest of SOMEONE ALWAYS MAKES PROGRESS (4.1.20). He gives the following as his rationale: "The difficulty is that you don't know what it is that you don't know. Only by making some movement can you detect what it is you don't know. Once you come to know what it is you don't know, you can pursue that information directly." And he adds an interesting admonition: "If you only 'clear the fog' and 'clear the fog' and 'clear the fog', you will not make real progress. You will have lots of little experiments and no deliverable results."

Bruce Whitenack's RAPPeL pattern language also presents a PROTOTYPES (A.5.23) pattern ([Whitenack 1995], p. 288). He adds the admonition:

> The dark side to prototyping is that solutions can be hacked together with the software inadequately robust and not well designed. It takes maturity, discipline, and a very good programming/design environment to reengineer quality back into a product. Without rigor and discipline a product is in serious risk of failure when features are continually added. As more prototyping and evaluating are done, there will be the need to modify the requirements. Iteration between prototyping and use-case modeling occurs during requirements analysis. In addition, user expectations have to [be] kept realistic as a prototype is not a product. Customers must realize that what they are seeing is a product simulation — not the product itself.

He also distinguishes between *low-fidelity prototypes* and *high-fidelity prototypes*:

> Work with the customer to build (initially) low-fidelity prototypes ... using paper widgets, drawings, self-stick notes, and index cards. (These are true throwaway prototypes.) Or, if the necessary skills and tools are available, build high-fidelity prototypes. (You do not want to spend more than 10 percent of your time on how to use the tool instead of focusing on the actual prototype, however.) Alternate between prototyping and use-case modeling. Prototyping provides more user involvement, and use case modeling provides rigorous analysis. Augment the use case documentation with references to prototype versions (product simulations).
>
> The high-fidelity prototypes that are developed with a tool capable of generating useful code may be used for evolutionary development. It may not be a throwaway prototype, but it should be developed in the spirit that it will be thrown away, which means making sure that all on the project—especially managers—understand that the prototype *may* be thrown away. It has been my experience with Smalltalk development that if developers have a good design in mind and if they are experienced, the prototype will probably contain code that is very usable for a production version. Be sure to plan for training of beta users and for creating a number of prototypes for prospective users.

Building and demonstrating prototypes is an art in itself. See the excellent pattern language DEMO PREP (A.5.9) by Todd Coram [Coram 1996] for guidance on the building, administration, and demonstration of prototypes. See also an earlier work by Ian Graham [Graham 1991].

The risk to your project of a small, throwaway effort is a small schedule delay. The risk of making a poor technical choice is a poor product or perhaps a commitment to a technology that simply will not work.

Be careful not to be seduced by the siren song of a successful prototype. Prototypes almost never can demonstrate capacity, reliability, or performance, which are often the most troublesome issues in development. The danger is that we see a prototype working and naturally assume that it will scale gracefully, run for weeks without rebooting, or perform nimbly under a customer's typical load. A working prototype does not imply that these problems are solved.

Contrast this pattern with SKUNKWORKS (4.2.14), which many think of as prototyping on a larger scale, but which is actually a little bit different in its forces and intent.

"The best friend of the architect is the pencil in the drafting room, and the sledgehammer on the job." — Frank Lloyd Wright, quoted in [Jacobs 1978].

4.1.8 Surrogate Customer

See section 4.2.7.

4.1.9 Take No Small Slips **

Boarding house, Washington, D.C., 1942, morning bathroom line. Small slips in the bathroom schedule build up, causing unfulfilled expectations downstream and leading to discomfort and dissatisfaction on the part of others.

Our project was in trouble. Everybody knew it. And then our project manager left the company. When our new project manager arrived, he called us all together. "I believe in taking one schedule slip," he said. Then he announced a 3-month slip. We all returned to work and redoubled our efforts. It was a challenge to meet the revised schedule, but he (and we) stuck to it, and we ultimately completed our development without incurring another slip.

... development is underway, and progress must be tracked, thus avoiding major surprises to both the customer and the enterprise.

It's difficult to know how long a project should take, and it's even more difficult to recover when the guess is wrong.

If you guess pessimistically, developers become complacent, and you miss market windows. If you guess optimistically, developers become burned out, and you also miss market windows. Projects without schedule motivation tend to go on forever or to spend too much time polishing details that are either irrelevant or that don't serve customer needs.

Therefore:

Prefer a single large slip to several small slips. ([Brooks 1995], page 24.)

As Paul Chisholm notes, "We found a good way to live by TAKE NO SMALL SLIPS from ... [Frederick Brooks'] *The Mythical Man-Month.* **Every week, measure how close the critical path (at least) of the schedule is doing. If it's 3 days behind schedule, track a 'delusion index' of 3 days. When the delusion index gets too ludicrous, then slip the schedule.** This helps avoid churning the schedule" [Chisholm 1994].

This pattern helps support a project with a flexible target date.

Dates are always difficult to estimate. DeMarco notes that one of the most serious signs that an organization in trouble is a schedule worked backward from an end date [DeMarco 1993].

A single large slip is important for the morale of the team. If you continually take small slips, nobody believes the schedule anymore, which hurts morale, reduces the sense of urgency, and encourages people to stop caring. On the other hand, a single large slip preserves at least some of the believability of the schedule, and people tend to be more willing to work toward a revised schedule.

Much of the rationale is supported in the MIT project management simulation, the Borland Quattro Pro for Windows case study, and Brooks' seminal work [Brooks 1995].

Most sane projects are managed this way.

See also RECOMMITMENT MEETING (4.1.12).

4.1.10 Completion Headroom **

Speaking of headroom ...

... work is progressing as the software unfolds and as the team learns more about the system from the customer and from the behavior of the system itself. Things are far enough along to start thinking about making a delivery to the customer on the agreed delivery date.

Every project must commit to delivery on a few hard and fast dates. This commitment is actually fortunate because it is about the only way to get out of work that is going poorly. It's also usually more important to deliver *something* on a specified date than to deliver everything at a later date: *When* is often more important than *what.* A WORK SPLIT (4.1.11) provides such a graceful exit by allowing the portion of work that is not understood or that is going poorly to be deferred while saving the part that does work or that will help the organization save face. A WORK SPLIT does require some advance notice since some portion of the work must still be completed by the deadline.

Therefore:

Project work group completion dates from remaining effort estimates in the WORK QUEUE REPORT [Cunningham 1996]. Compare the largest of the earliest completion dates for

each work group to any hard delivery date that may apply. The difference is your COMPLETION HEADROOM.

Any group has an obligation to make their efforts visible through what becomes the ultimate trouble signal, low COMPLETION HEADROOM. Headroom disappears when developmental activities fail to match those of COMPARABLE WORK [Cunningham 1996].

In order for COMPLETION HEADROOM to work, it must be calculated from the beginning and recalculated often, at least weekly. Watch for trends. Headroom will often shift plus or minus a day or two from week to week. But steady evaporation of headroom for any WORK GROUP (A.5.30) is a sure indicator that management attention is needed. You can reorder the WORK QUEUE (4.1.13), possibly defer entire items to a later release, use the WORK SPLIT (4.1.11) pattern already mentioned, or face the public embarrassment of a RECOMMITMENT MEETING (4.1.12).

A common problem is the well-meaning escalation of requirements by people who are too close to a problem. If you track COMPLETION HEADROOM, you are in a better position to assess the impact of adding these requirements to the project.

See also TAKE NO SMALL SLIPS (4.1.9).

A version of this pattern first appeared in [Cunningham 1996].

4.1.11 Work Split *

... a WORK GROUP (A.5.30) commits to resolve and deliver IMPLIED REQUIREMENTS (4.1.16) in the most timely and satisfactory way they can find. They are not committed to specific dates.

A work group has an obligation to make its efforts visible through what becomes the ultimate trouble signal, low COMPLETION HEADROOM (4.1.10). Headroom disappears when developmental activities fail to match those of COMPARABLE WORK. A common problem is the well-meaning escalation of requirements by people who are too close to a problem.

Therefore:

Divide a task into an urgent component and a deferred component such that no more than half of the developmental work is in the urgent half. Defer more work, if required, to acquire sufficient COMPLETION HEADROOM (4.1.10). Defer analysis and design of parts that won't be implemented (this advice runs counter to conventional wisdom).

Often, a split is just a way to get back to the basic work that had been originally planned. Trust architecture and requirements substitutions to cover for omissions and inconveniences caused by incomplete up-front work. Both halves of the split will appear in the WORK QUEUE (4.1.13) with distinctly different urgency levels.

The split should be based on clear business priorities or should otherwise be rooted in agreed values. Ian Graham has written patterns that combine to form a small pattern language (drawn from a larger pattern language) to address this issue. See the patlets for BUSINESS PROCESS MODEL (A.5.6), ESTABLISH THE BUSINESS OBJECTIVES (A.5.12), and GRADUAL STIFFENING (A.5.14).

A version of this pattern first appeared in [Cunningham 1996].

4.1.12 Recommitment Meeting *

... each development group is managing its schedule using WORK SPLIT (4.1.11), but additional scheduling problems seem to keep coming up.

If a product initiative is in jeopardy because IMPLIED REQUIREMENTS (4.1.16) cannot be met through schedule and WORK QUEUE (4.1.13) adjustments, then it is unlikely that any other development-initiated activity will help. Management up to at least the level that began the initiative will suddenly take an interest in all circumstances leading up to the current situation. Some of this analysis is natural and appropriate. However, this period won't be a time of high productivity, and it shouldn't be allowed to continue too long.

Therefore:

Assemble a meeting of interested management and key development people. In the meeting, review the history of the situation until all present agree that simple adjustments (e.g., working weekends or adding staff) won't help. Eventually a solution appears, usually expressed as a question of the form: What is the least amount of work required to do X? (X is one person's idea of the most important part of the initiative.) The question should be answered quickly and confidently by consulting a recent WORK QUEUE REPORT [Cunningham 1996].

The process may repeat for plans Y and Z. Ultimately, a plan will be selected. Then, the remainder of the meeting is devoted to talking through the implications of the decision and getting all parties' commitment to the new plan and/or schedule.

This pattern, of course, is another form of episode. The decisions are ones of business resource allocation and thus belong in upper management. However, all present can and should contribute in a frank, honest, nondefensive, and constructive way.

See also TAKE NO SMALL SLIPS (4.1.9).

A version of this pattern first appeared in [Cunningham 1996].

4.1.13 Work Queue *

... IMPLIED REQUIREMENTS (4.1.16) suggest deliverable program enhancements that will have various necessities, dependencies, risks, and rewards. Deliverables may be ill-defined, being represented more by a vision or desire than by anything concrete or measurable.

It is difficult to perform linear, monochronic scheduling in light of IMPLIED REQUIREMENTS (4.1.16).

If we were to work up a conventional schedule, we would probably begin with a block of requirements analysis for each item. From these blocks would be hung blocks of specification, design, implementation, and eventually integration and testing. Add to this structure some wild guesses and a few ordering constraints, and, presto, you have a 30-foot diagram showing what will be finished when and by whom. Such a document takes on a life of its own, striking fear in developers' hearts and generally distracting everyone else from the real scheduling task, which is to get better input, not larger output.

Therefore:

Produce a schedule that is simply a prioritized list of work. Use the list of IMPLIED REQUIREMENTS (4.1.16) (really just names) as a starting point and situate them into a likely implementation order, favoring the more urgent or higher priority items. When work can

be factored from two or more entries, go ahead and do so, giving the common element a name that establishes its worth and implies its implementation precedence.

EXAMPLE:

1. Settlement-Data Positions
2. Settlement-Date-Based Tax Reports
3. Trade vs. Settlement Accounting Preference by Portfolio

Be prepared to reorder this list as unforeseen interactions surface or as business realities demand new priorities. Remove work from the list as that work is completed. Observed defects are not enough to return completed work to the list. However, independently scheduled repair activity may uncover omissions that are more appropriately removed from defect tracking and scheduled in competition with all of the other work in the WORK QUEUE (4.1.13).

A version of this pattern first appeared in [Cunningham 1996]. The pattern is similar to the later SCRUM pattern BACKLOG ([Beedle 1999], p. 643–644), which is summarized in ([Rising 2000], p. 146):

> To organize the work remaining on a project, maintain a prioritized list, the Backlog. The list is dynamic and updated at the end of each Sprint.

4.1.14 Informal Labor Plan **

Workers using an informal labor plan to construct an adobe building, Penasco, New Mexico.

We were discussing the introduction of new project management software. One project manager protested that it didn't provide the granularity she needed. It turned out that she wanted to track items that were fractions of days of effort.

... real development requires developers to work on several parallel tasks such as DEVELOPMENT EPISODES (4.1.15) that may have interdependent or even conflicting priorities and due dates.

A schedule of developer work tasks can both assist workers in planning their time and provide reassurance to stakeholders about scheduling expectations. The DEVELOPMENT EPISODE (4.1.15) presents an ideal that must be worked into the lives of people trying to get a big job done quickly. Developers often find themselves obligated to work on more than one in-progress DEVELOPMENT EPISODE at a time. The WORK QUEUE (4.1.13) offers one prioritizing method, though it ignores the many small trade-offs possible when the work is at hand.

Therefore:

Let individuals devise their own short-term plans. Accept that much of the group activity implied in a DEVELOPMENT EPISODE will take place between group members who find the time to tackle some issue together [DEVELOPING IN PAIRS (4.2.28)]. Avoid the temptation to

call a meeting where a developmental climax is intended to happen. It won't. Instead, let individuals express interests and make commitments to each other. And let them revise these intentions on a moment's notice when the energy of some episode reaches an irresistible level.

Accordingly, there is a threshold of detail below which a project manager should not track. The threshold may vary depending on the project, but it is a safe bet that tasks that require less than a few days of effort should not be formally tracked. One might get a sense of excess detail by the amount of developer complaints about the relevance of the tracking.

The above mentioned practices lead to an organization where the DEVELOPER CONTROLS PROCESS (4.1.17). Not only does the developer suggest the overall structure of commitments, but the developer also becomes the focal point for day-to-day priority calls.

A DEVELOPMENT EPISODE is actually composed of a series of PROGRAMMING EPISODES (4.1.19), some of which must take place in (at least) pairs if any approximation of group consciousness is to form. Individuals' labor plans are the tools they use to make these connections happen. Pair programming facilities [Beck 1999] are configurations of the physical environment that can reduce this planning to an occasional HALLWAY CHATTER (5.1.15) promise.

A version of this pattern first appeared in [Cunningham 1996].

4.1.15 Development Episode *

A baseball game is divided into separate episodes, called innings.

... members of a WORK GROUP (A.5.30) have been selected based on needs inferred from the IMPLIED REQUIREMENTS (4.1.16).

It's important to build on the collective strength of an entire team and to build a true gestalt from the team members.

Each team member contributes specific skills that will be important at some point in the development. For this we can be thankful. However, if we overemphasize a team member's specific strength, we diminish the perception of everyone's general abilities, unnecessarily narrow the team member's focus to the application of that specialty, risk creating ambiguity as to who is responsible for nonspecialized tasks, and discourage the learning of new skills.

Therefore:

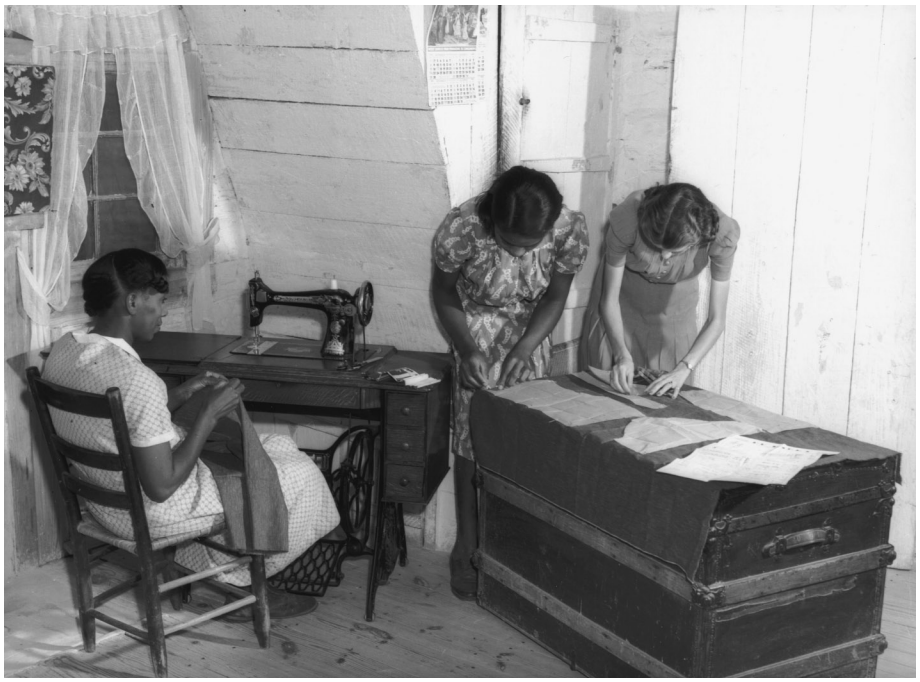
Approach all development as a group activity as if no one had anything else to do. Expect the activity to follow the usual course of an episode, where energy builds to a decision-making climax and then dissipates. At the height of the episode, purpose should be clear, terminology well understood, knowns well explored, and unknowns identified. It is at

exactly this point that individual strengths merge into a sort of common consciousness. Landmark decisions come easily. Breakthroughs are common. A creative act will have been shared.

Besides yielding better decisions, the collective episode has very positive effects on the participants. Looking back, people often have trouble identifying the actual source of key ideas. Nonspecialists gain invaluable insight into the thought processes of the specialist, whose ideas are demystified and shared throughout the group. Specialists will realize that this sharing will not diminish their own status within the group. The specialist may even delay sharing some insights, realizing that their actual recognition experience will be of tremendous value to the nonspecialists and a small loss to himself or herself. Seymour Papert called this an "Ah Ha" moment and admonished instructors not to "Steal the Ah Ha" [Papert 1980].

A version of this pattern first appeared in [Cunningham 1996].

4.1.16 Implied Requirements



Farm Security Administration (FSA) home supervisor Miss Harton helping some members of a borrower's family cut patterns and make their own clothes. Caswell County, North Carolina. Pattern parts such as sleeves are chunks of functionality that are well understood by the customer.

... a PRODUCT INITIATIVE (A.5.22) has identified the direction for further development, and a MARKET WALK-THROUGH (A.5.16) has explored the customer motivation and developmental possibilities behind that initiative. We expect positions and attitudes to be understood, but we have yet to make any commitments beyond everyone's general commitment to do a good job for the company.

✣ ✣ ✣

A commitment implies an agreement between people. Development commitments generally obligate developers to meet some customer need in a timely and satisfactory way. The tension here is to define a need in sufficient detail so that commitments have meaning without exhausting up-front analysis or overconstraining a solution.

Therefore:

Select and name chunks of functionality. Use names that have meaning to customers and that are consistent with the PRODUCT INITIATIVE (A.5.22). Allow these names to imply customer requirements without actually enumerating requirements in the traditional sense.

EXAMPLES:

- Year-End Tax Reports
- Dollar-Denominated Japanese Bonds
- High-Quality Printing
- Disconnected Operation on Laptops

These names will fill in the blank in recurring questions like the following: Who's handling the programming (or specification, or customer contact, or manual update, or release notes) for _____.

See also NAMED STABLE BASES (4.1.4).

A version of this pattern first appeared in [Cunningham 1996].

4.1.17 Developer Controls Process **

A journeyman devises effective and efficient processes for the manufacture of self-sealing fuel tanks during World War II.

... an organization has come together to build software for a new market in an immature domain or in a domain that is unfamiliar to the development team. Progress will be marked by an INFORMAL LABOR PLAN (4.1.14). The necessary roles have been defined and initially staffed.

A development culture, like any culture, can benefit from recognizing a focal point of project direction and communication. Successful organizations work in an organic way with a minimum of centralized control. Yet important points of focus, embodied in roles, tie together ideas, requirements, and constraints into an artifact ready for testing, packaging, marketing, and delivery.

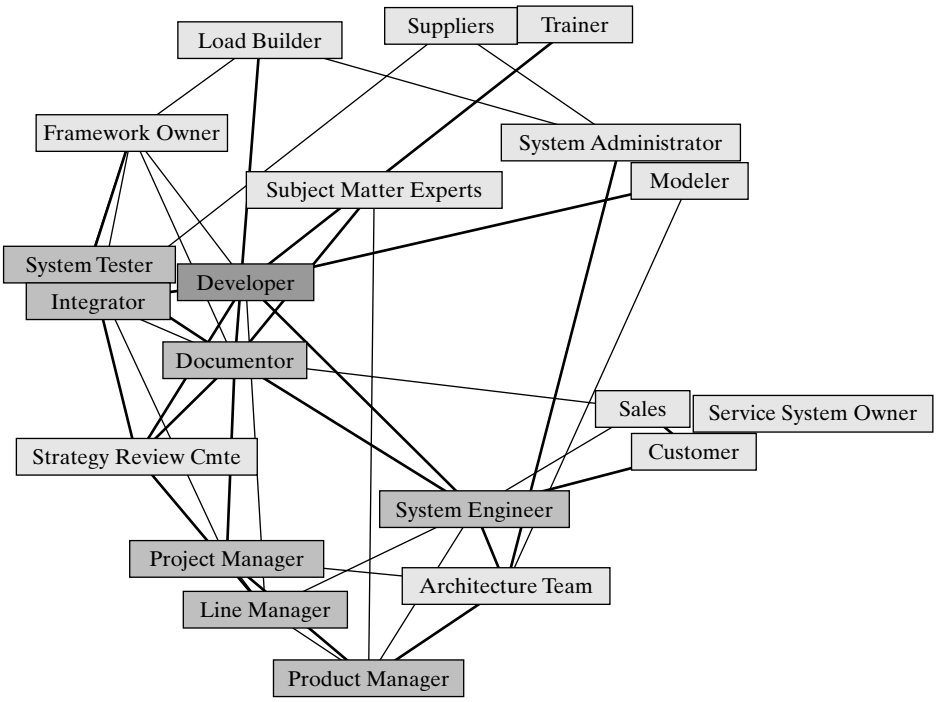
Strict control is viewed by most development teams as a draconian measure. The right information must flow through the right roles. You need to support information flow across analysis, design, and implementation.

Because developers contribute directly to the end-user-visible artifact, they are in the best position to take accountability for the product. Of all roles, they have the largest stake in the

largest number of phases of product development. And there should be no accountability without control. The manager has some accountability as well, to the extent that he or she indirectly supports delivery of the user-visible artifacts. These are process issues.

Therefore:

Make the Developer the focal point of process information. In the spirit of ORGANIZATION FOLLOWS MARKET (5.1.9) 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. Responsibilities of developers include understanding requirements, reviewing the solution structure and algorithm with peers, building the implementation, and performing unit testing.



The developer is central to all activities of this end-to-end software development process.

Note that other hubs, such as a Manager role, may exist as well, though they are less central than the Developer role.

The Developer who is at the hub of a particular feature may be accorded that position according to FEATURE ASSIGNMENT (5.2.14), but, more generally developers should be at the communication hub of whatever process engages them in writing code for the customer. This pattern

encourages a structure that supports its prime information consumer. The Developer can be moved toward the center of the process using the patterns WORK FLOWS INWARD (4.1.18) and MOVE RESPONSIBILITIES (5.1.18).

Though Developer should be a key role, care must be taken not to overburden that role. This pattern should be balanced with MERCENARY ANALYST (4.1.24), FIREWALLS (4.2.9), GATEKEEPER (4.2.10), and more general load-balancing patterns like RESPONSIBILITIES ENGAGE (5.1.14), HALLWAY CHATTER (5.1.15), and MOVE RESPONSIBILITIES (5.1.18). The Developer should enjoy particularly strong support from the PATRON ROLE (4.2.15), and conflicts can be escalated to the PATRON ROLE when consensus breaks down.

If the Developer controls the process, then it's possible to implement the pattern WORK FLOWS INWARD (4.1.18).

Developers, of course, don't control the process unilaterally, but as a collective group, starting with DEVELOPING IN PAIRS (4.2.28).

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. [In the figure, the Architect role is split across Framework Owner and ARCHITECTURE TEAM (5.2.4).] 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.

In a mature domain, consider HUB SPOKE AND RIM (5.1.17) as an alternative.

You can still write down your process as part of a process improvement program. But keep the documentation light; many organizations have found that one page per process is good enough. And make sure each process step meets a need that you can tie to your organization's value proposition. Most often, this value is or should be tied to the product you are producing for a paying customer. If it isn't obvious how the process step helps to achieve what you know the customer wants, then do the right thing instead.

4.1.18 Work Flows Inward **

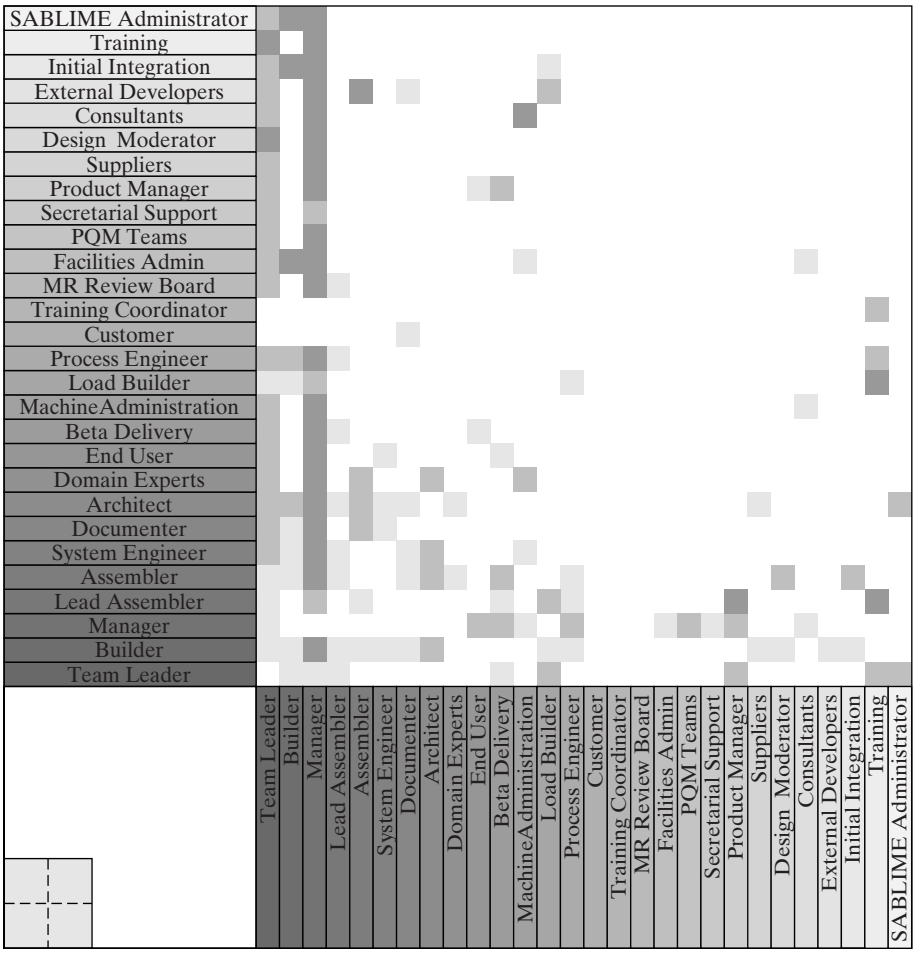
Work (i.e., pears) flowing into a pear processing plant.

... an organization is in place and has been doing work long enough that it can introspect about its structure and workings. There is some management pecking order or hierarchical decision-making structure in the organizational network. Work instructions flow through this structure, with the possibility that each role makes decisions, adds constraints, or works to carry out decisions within some set of constraints.

An organization must seek a structure that best ensures that the most authoritative roles make the decisions and carry out the work that adds value directly to the product.

Some centralized control and direction are necessary. During software production, the work bottleneck of a system should be at the center of its communication and control structure. If 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 (4.2.9) and GATEKEEPER (4.2.10) (no centralized role need fill this function).

Look at the following grid that depicts the directed flow of communication in an organization [see How the Patterns Came to Us (Chapter 2)]. In this organization, a core of roles at the center initiates interactions across the spectrum of most of the other roles.



Yet, this core receives very little input from the rest of the roles in the organization, and this core is rife with management roles (Team Leader, Manager, Lead Assembler). It has an overloaded center, and work requests flow outward from this center, diffusing across the other roles. Core roles *make* work.

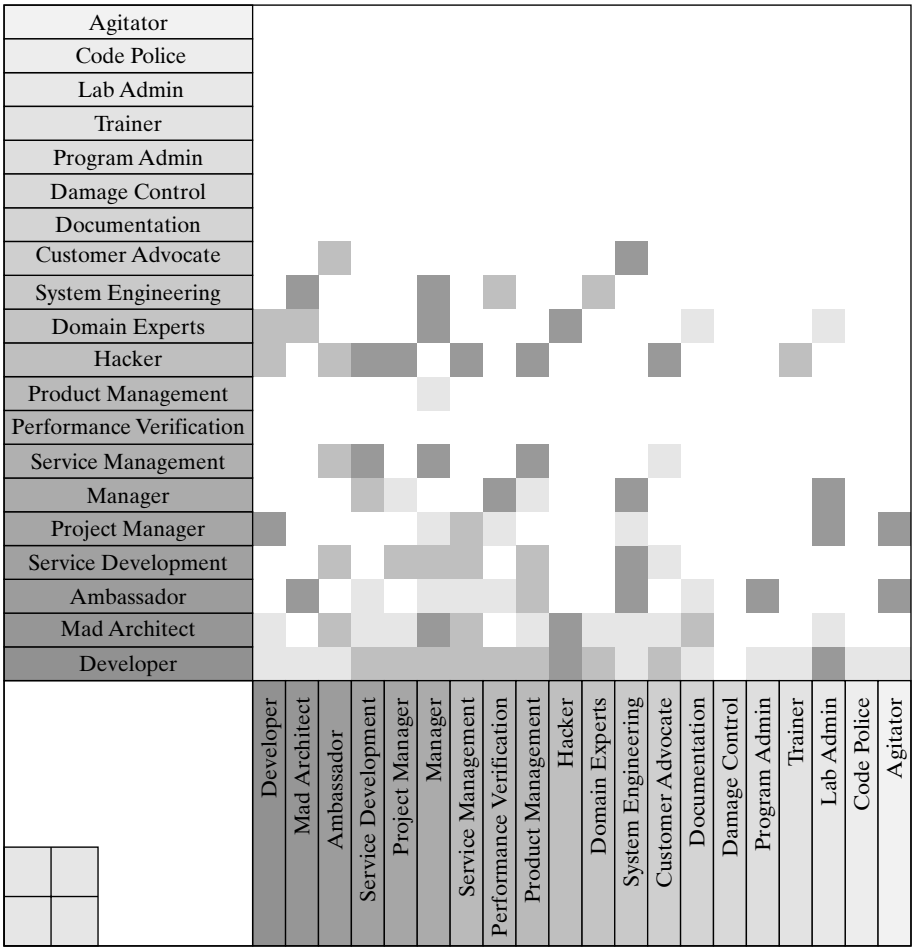
Katz and Kahn's analysis of organizations shows that the exercise of control is not a zero-sum game ([Katz Kahn 1978], p. 314).

Therefore:

Work should flow in to Developers from stakeholders, especially customers. Work should not flow out from managers.

Managers should not be situated at the center of the communication grid: They will become overloaded and make decisions that don't take day-to-day dynamics into account.

Consider the following picture, where work flows from the roles across the organization to the roles at the center: Developer, Architect, and Ambassador. The grid shows a healthy distribution of inward-directed inputs. And, in large part, the central roles *do* work, rather than *make* work.

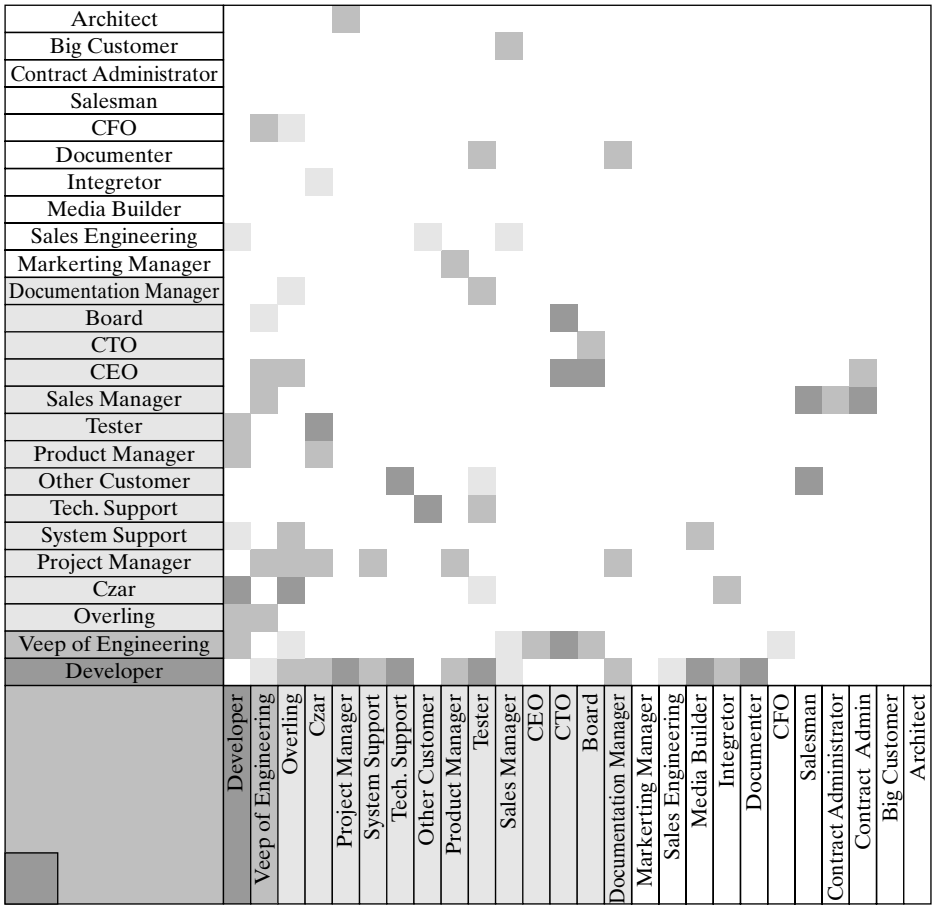


✣ ✣ ✣

The result is an organization whose communication grid has more points below the diagonal than above it (as in the figure above).

The work should focus at the center of the process, and the center of the process should focus on value-added activities [DEVELOPER CONTROLS PROCESS (4.1.17)].

But consider this interaction grid:



Superficially, the graph appears to show a WORK FLOWS INWARD pattern. But, in fact, most of the interactions directed from outlying roles to the developer are of an imperative nature rather than of an informative nature. The developer role is being pulled in many directions, and the organization health suffers greatly.

Organizations run by professional managers tend to have repeatable business processes, but don't seem to reach the same productivity plateaus as organizations run by engineers. In programmer-centric organizations, the value-added roles are at the center of the process [DEVELOPER CONTROLS PROCESS (4.1.17) and ARCHITECT ALSO IMPLEMENTS (5.2.10)]. The manager should facilitate and support these roles and their work [FIREWALLS (4.2.9) and PATRON ROLE (4.2.15)].

Mackenzie characterizes this pattern using *M-curves*, which model the percentage of task processes of each task process law level (planning, directing, and execution) as a function of the classification [Mackenzie 1986].

The rationale is supported with empirical observations from existing projects.

The broad goal of this pattern is to separate overhead work from central work. DAY CARE (4.1.23) is another pattern with a similar intent.

The Manager should still make day-to-day decisions for the business process and accept the responsibility to "keep the pests away" [FIREWALLS (4.2.9)].

In his new work *The Nature of Order,* Christopher Alexander speaks of *gradients* as one of the 15 structural properties of whole systems that emerge naturally in a process of local adaptation [Alexander 2003]. In WORK FLOWS INWARD, there should be a natural gradient of information flow toward the developer at the "center" of the organization—both in the sense of the social network diagrams and in the sense that Alexander uses the term "center" to describe a prominent feature of a system.

4.1.19 Programming Episodes **

Making the possible decision now: What kind of candy can I buy with my nickel?

... you have a good idea of where to start and perhaps even some fledgling pieces of code. Now you need to establish a rhythm of productive development that can engage and fuel the team.

✣ ✣ ✣

Programming is the act of deciding now what will happen in the future, but it always seems like some parts of the future don't happen soon enough and other parts are always too far off and out of reach. A programming language offers an operationally precise way to encode decisions through a process called coding. Programmers reason about future behavior by interpreting previously coded decisions and integrating these decisions with their own decisions, their interpretations of other sources like technical memos, and the guidance of domain experts. The depth, quality, and value of programming decisions will be limited by the programmers' ability to concentrate.

Therefore:

Develop a program in discrete episodes. Select appropriate deliverables for each episode and commit sufficient mind share to complete these deliverables by making the possible decisions now and coding those decisions. Be aware of the rise in concentration as the episode progresses. Consider each source and consciously include or exclude its recommendations.

✣ ✣ ✣

Fear often accompanies a decision that has not yet been made. Use this fear as a motivation. Try to compare your position within an episode with similar points in previously successful episodes.

EXAMPLE:

"I feel like we've been around twice now on the possible ways we can bind the six terms of this bond analytic to the four calculation classes we have in our library."

"Yeah, right now I'd be happy if we could place the four primary terms, look at the error cases, and see if all that gives us a hint as to how to proceed after lunch."

Push for the decisions that can be made. Don't abandon an episode; doing so will leave you feeling defeated and unable to achieve even the same level of concentration at a future time. Make the decisions that seem possible, code the decisions, and then review the code to be sure that the extent of your decisions and your confidence in them are apparent in the code. Coding occurs on the downhill side of a programming episode. Coding is the most direct way to promulgate programming decisions.

A version of this pattern first appeared in [Cunningham 1996].

4.1.20 Someone Always Makes Progress*

Room enough for everyone to work ...

... secondary tasks are dominating the team's time, keeping them from moving forward with their primary goal. There are common complaints about distractions.

✣ ✣ ✣

It is important to keep a team moving forward and to avoid getting stuck on the obstacles. You need to pay attention to every task, including small diverting ones. But you also need to complete the primary task by an important date.

Therefore:

Ensure that someone on the team is making progress on the primary task at all times.

✣ ✣ ✣

If you do not complete your primary task, nothing else will matter. Therefore, complete that task at all costs.

You can employ one of a broad range of particular solutions and tactics depending on the exact forces to be resolved. The following specializations are example refinements of this pattern:

- DEVELOPING IN PAIRS (4.2.28)—allow one person to take the keyboard.
- TEAM PER TASK (4.1.21)—separate tasks into sympathetic sets.

- SACRIFICE ONE PERSON (4.1.22)—assign only one person to the distraction.
- DAY CARE (4.1.23)—separate the training task from the task of producing software.

But, in any case, making progress on the primary task will always bring you closer to your final goal, which is not always the case when dealing with distractions.

The psychological effect of this pattern should not be underestimated. If the project is hit with many distractions, it can be demoralizing to see work grind to a halt. However, any visible progress will help the entire team stay focused and will encourage them to get through the crisis so that they too can once again make progress.

Carried too far, this pattern might lead you into trouble for not adequately addressing the distractions. However, too many distractions are usually a symptom of some other problem [see, for example, FIREWALLS (4.2.9)].

SAMPLE SITUATIONS:

A. Scylla and Charybdis. In The Odyssey, Odysseus has to sail past either Scylla or Charybdis. If Odysseus chooses to sail past Scylla, a six-headed monster, six of his crew members will be eaten, but the rest will survive. If Odysseus chooses to sail past Charydbis, a whirlpool, the entire ship will be destroyed. In this paradigmatic dilemma, Odysseus chooses to sacrifice six people rather than sacrifice his entire crew.

B. Atalanta. In the Greek story of Atalanta, Atalanta is assured by the gods that she will remain the fastest runner as long as she remains a virgin. So she tells her father, the king, that she will only marry the man who can beat her in a foot race; the losers are to be killed for wasting her time. The successful young man is aided by a god, who gives him three golden apples. Each time Atalanta pulls ahead, he tosses an apple in front of her. When she pauses to pick up each golden apple, he races ahead and eventually wins the race.

The moral of this story is that Atalanta should not have stopped to pick up the apples, which also illustrates the point of this pattern. I choose to view the story metaphorically; Atalanta represents distractions trying to beat you to your project's deadline. The apples are members of your team, whom you will separate from the main team one at a time to ensure success.

See [Csikszentmihalyi 1990] and [DeMarco Lister 1976].

A version of this pattern first appeared in [Cockburn 1998].

4.1.21 Team Per Task **

... a big diversion hits the team, threatening to disrupt ongoing work and temporarily halt progress.

Large distractions (usually called crises) must not be allowed to stop a project, even for a short time. Crises are inevitable, and they can occur frequently. If the project members take time to respond to each crisis, they will soon find themselves spending so much time performing crisis management that the real work doesn't get done.

Of course, the diversions are real. A previous release needs an emergency bug fix. New people must be trained. The ISO audit will happen. But these diversions must be handled in such a way that the project still moves forward.

The temptation is to throw everything you have at these high-priority items and to let the whole team work on the issues until they go away. However, such an approach confuses urgency with amount of effort. Some problems require only a small amount of attention, although that attention should be *immediate*. A stitch in time saves nine.

Therefore:

Let a subteam handle the diversion, which allows the main team to keep working.

One approach is to split the team. Sort the activities so that each team has a primary task with additional, sympathetic activities. Sitting in meetings, answering phone calls, and writing reports, for example, are nonsympathetic to designing software. Arrange it so that each team can focus on its primary task and so that each task has at least one dedicated team member.

As a result, important distractions are handled almost entirely by specialized teams, thus allowing the main team to continue uninterrupted.

However, one must be careful not to overdo it. Carried to extremes, this pattern results in single-person teams. In addition, while solving a crisis is important, be careful not to heap praise too lavishly on the crisis teams. Otherwise, crisis management becomes the glamor job, and the team focuses on putting out fires rather than on building the building. [See COMPENSATE SUCCESS (4.2.25).]

PRINCIPLES INVOLVED:

Increase flow time and decrease distractions, thus trading personnel parallelism for time-slicing. *Flow* is the quiet time in the brain when the problem flows through the designer ([Csikszentmihalyi 1990] and [DeMarco Lister 1976]). It is when design alternatives are weighed and decisions are made in rapid succession as mental doors open. The problem, the alternatives, and the state of the decision process are all kept in the head. It is a not only a highly productive time, but it is also the only time when the designer feels comfortable making decisions.

It takes about 20 minutes to reach the internal state of flow and only a minute to lose it. Beyond getting into the flow, the designer must have time to actually make progress, which may take another 10 minutes. Any significant interruption within that half hour essentially causes the entire half hour to be lost. As it takes energy to get into the flow, so too a distraction costs energy as well as time.

To increase flow time, distractions have to be reduced. Certain pairs of activities are more mutually distracting that others. Fixing a bug requires flow in the old system and hence distracts from flow in the new system. Sitting in meetings, answering questions, and time on the telephone are major distracters to design flow. Therefore, the recommendation is to group tasks into sympathetic sets. Requirements and analysis involve attending meetings, reading, and writing. Design and programming require concentration on the implementation technology and the ability to keep a great number of details in the head.

Note that time-slicing, whereby each person will do design some part of the time, can be more attractive in terms of job satisfaction. The significant time needed to switch between tasks causes parallelism to be preferred in this case. Some of the people may adopt the new task as their profession [see SACRIFICE ONE PERSON (4.1.22), DAY CARE (4.1.23), and FIREWALLS (4.2.9)].

EXAMPLE:

Concurrent gathering of requirements and designing of software:

Project Winifred tried having each person gather requirements and perform analysis, design, and programming. We thought that the developers would enjoy the variation in activities and that the developers' multi-tasking would reduce the meetings and bureaucratic documentation exchanged between people.

What happened, however, was that the first two activities were so different from the latter two that people were unable to switch easily between them. After attending meetings and writing documentation for much of the day, people found it difficult to start working on design and programming. As with bug fixing/new development, every time a designer was pulled away from her or his work, it cost an additional hour to recover the train of thought.

We applied TEAM PER TASK and split the teams along task lines. Requirements gathering and analysis were assigned to designated people in each team, and design and programming were assigned to others. The result was that the requirements/analysis people sat in meetings, read and wrote specifications, examined interfaces, etc. They communicated their findings to the designers/programmers—orally, for the most part, since they were closely linked on the same team [HOLISTIC DIVERSITY (4.2.19)]. The designers/programmers stayed in their train of thought, getting fresh input from their requirements colleagues. Some of the people assigned to develop requirements really wanted to program, so their assignment was quite a sacrifice for them [SACRIFICE ONE PERSON (4.1.22)].

There are two things we did not do. We did not put the requirements/analysis people into a separate team [HOLISTIC DIVERSITY (4.2.19)]. Instead, each team was jointly responsible for a section of the system, from requirements to delivery. The splitting thus occurred within each team. We also did not force the requirements group to document their decisions for the designers' benefit (they did document their decisions for the project's benefit). The requirements and design people were in close contact at all times, and most information was communicated orally. There was, therefore, no "throw it over the wall" effect. These important teaming decisions were made earlier, and we were intent on preserving them.

RELATED PATTERNS:

This pattern treats each task both as an activity and as a deliverable.

OWNER PER DELIVERABLE (A.5.19)—addresses the general form of ownership and accountability.

FUNCTION OWNER AND COMPONENT OWNER—establishes a team for each artifact, and addresses the task of designing each artifact.

SOMEONE ALWAYS MAKES PROGRESS (4.1.20)—addresses the general distraction management pattern.

SACRIFICE ONE PERSON (4.1.22)—addresses specialization to lose only one person.

DAY CARE (4.1.23)—addresses training as a separate deliverable from the software.

SACRIFICE ONE PERSON (4.1.22).

READING:

See [Csikszentmihalyi 1990] and [DeMarco Lister 1976].

A version of this pattern first appeared in [Cockburn 1998].

4.1.22 Sacrifice One Person *

Alias: SACRIFICIAL LAMB

... during a typical project, there are always a host of small distractions.

Small distractions can add up and sap the strength of the team.

Even small distractions must be handled, but it is important to note that they take time away from the primary task. In particular, any distraction, even a small one, disrupts "flow" time, which costs significant additional time to regain.

Many small distractions involve less desirable jobs.

Therefore:

Assign just one person to the distraction until it is resolved.

This pattern is very much like TEAM PER TASK (4.1.21), except that the distraction is smaller. As such, it can seemingly be handled by one person half time to full time.

All but one member of the team moves forward distraction free. The person assigned to the distracting task may be unhappy, so try to get that person back on the team again as soon as possible. If you feel that one person is too much to sacrifice to this task and want to make it part time work, estimate the loss of flow time that would result from trying to address both this distraction and some other task.

If distractions keep happening, you will be left with no one performing the primary task, and you will need to examine why you have so many distractions in the first place.

OWNER PER DELIVERABLE (A.5.19) is the general ownership and accountability pattern. SOMEONE ALWAYS MAKES PROGRESS (4.1.20) is the general distraction management pattern. TEAM PER TASK (4.1.21) is the general form of this pattern at the team level.

Several patterns refine this pattern for specific contexts. DAY CARE (4.1.23) addresses training as a separate deliverable from the software and produces *mentor* as a profession. In FIREWALLS (4.2.9), the distraction is a series of requests from outside the team, so one of the developers is sacrificed to act as project manager (which can produce *project manager* as a profession). The MERCENARY ANALYST (4.1.24), usually a "hired gun," handles the distraction of documentation, leading to *technical writing* as a profession. And in GATEKEEPER (4.2.10), the constant inflow of technical information is the distraction, and one person is assigned to manage that information as a distinct, part-time task. GATEKEEPER is one of the major foundations for *manager* as a profession.

Don't forget the sacrificial lamb when it comes time to COMPENSATE SUCCESS (4.2.25).

PRINCIPLES INVOLVED:

As in TEAM PER TASK (4.1.21), the fact that handling the distraction looks like less than a full-time job illustrates the significance of the time spent getting into mental flow.

Maximum parallelism, profession, or sacrifice? If the people do not like the task, they consider it a sacrifice. If they like the task, it becomes their profession. Thus, FIREWALLS gives rise to the profession of project management, and DAY CARE gives rise to the profession of mentor.

EXAMPLES:

A. Updating the project schedule. On Project Winifred, the schedule was out of date. We thought it would be fair to let everyone on each team evaluate their own work in order to spread the experience, discomfort, and load. What really happened was that progress came to a total halt. When the design team got back to designing, a month had gone by with no design progress, and they had forgotten some of the design issues that had been in their heads. One of the teams used SACRIFICE ONE PERSON. They drew lots to choose one person to complete the whole team's estimation while the others got on with the main task. At the end of several weeks of estimation, that team had moved forward while the other teams were at a standstill. Thereafter, every team applied the pattern. The person working on the schedule really felt as if a sacrifice had been made. This pattern was originally called "Scylla," as described in the story of Scylla and Charybdis.

B. Simultaneous release to QA and development of the next release.

Project Winifred had one unit entering testing at the same time design was starting on the next. We optimistically thought the bug fixes would take a relatively small amount of time, and so we assigned the whole team to both fix bugs and perform new design.

Each fix broke a designer's train of thought for a period of time on the order of an hour, even though the fix itself took little time. Three or four bug fixes a day caused the designer to

lose most of the day. Eventually, the designers gave up on the new release because they knew the next bug fix would arrive before they would have recovered their thoughts and made progress on the new design.

We applied SACRIFICE ONE PERSON, and assigned one person to fix bugs. We originally planned it as a half-time job, but found there was not enough time left over for the person to do any useful design. The person rejoined the new design team as soon as the release went through testing.

A version of this pattern first appeared in [Cockburn 1998].

4.1.23 Day Care *

Alias: PROGRESS TEAM/TRAINING TEAM

.. the project has just brought on several new people.

Your experts are spending all of their time mentoring novices.

You begin to hear things like "We are wasting our experts," or "A few experts could complete the whole project faster." Indeed, the experts are not proceeding at the rate you or they would expect because training the new people is draining their energy, time, and concentration. But the new people must be trained, by experts, of course.

At the same time, you must make progress on the project itself.

Therefore:

Put one expert in charge of all of the novices, and let the others develop the system.

Separate an experts-only "progress" team from a training team under the tutelage of one or more mentors. Select the mentors for their ability to teach design and programming (e.g., object-oriented design and programming) to novices. Let the progress team design 85-95 percent of the system, and let the training team deliver only 5-15 percent of the system and instead focus on

quality training. Transfer people to the progress team as they become able to contribute meaningfully.

Make sure that the training team does not simply perform training exercises, but that they actually contribute to the final system in an ever-increasing way.

If you have many people to train (more than, say, six), you will have to design a series of tasks for them to attempt. Otherwise, you may give them a small, real part of the main system to design.

If the people in the training team are the ones who know the domain, you will have to make some further adjustment, or else the division of labor may cause conflict.

The result is that most of the experts can continue to make progress on the project. The novices contribute to a small part of the project that grows as they gain experience.

In extreme cases, though, you eventually have too few people to constitute a progress team.

How many people can one mentor train if the mentor performs training full time? A reasonable number is five. I have, however, heard of one person mentoring 15 people on five concurrent mini-projects.

PRINCIPLES INVOLVED:

The principles are synergy vs. distraction, the synergy of having a novice learn directly from an expert vs. the distraction experienced by the expert. Experts who have to answer novice questions are reduced to a fraction of their productivity, without particularly raising the productivity of the newcomers. Assigning one novice to work with an expert may cut the expert's productivity in half, assigning two may cut it to a third, and adding three may eliminate productivity altogether.

Assume X experts work at productivity 1 each and that a larger number of N novices work at n productivity each, with n being much smaller than 1 (on the order of 1/10 the productivity of the experts). If the experts could work together, they would have, in this simple model, a total productivity of (X).

If one expert is sacrificed to train the novices full time, that person has zero productivity. The group's total productivity using DAY CARE is

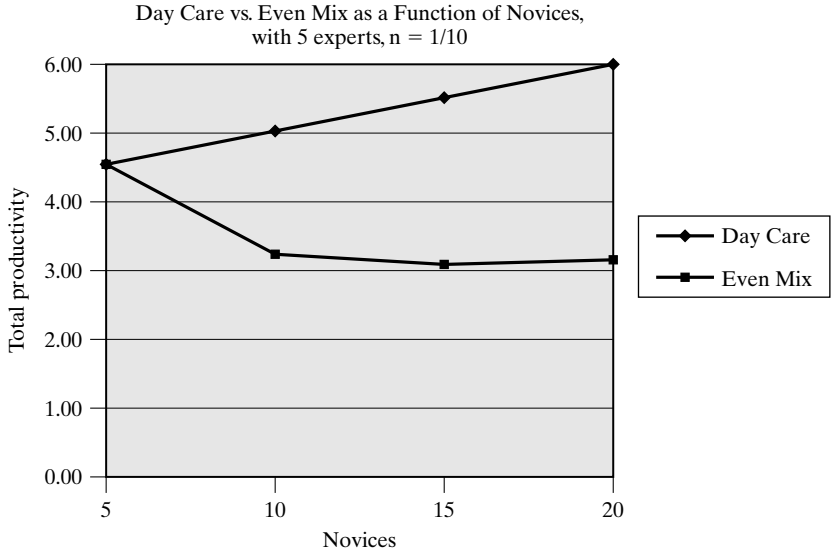
$$(X-1) + N*n$$

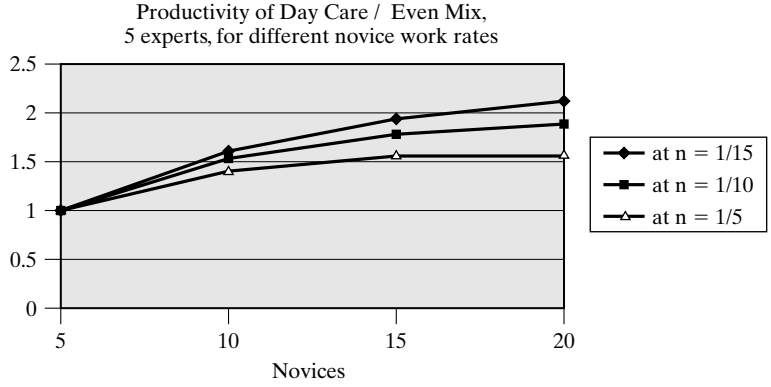
which is shown in the upper curve in the figure. If the experts and the novices are all mixed together ("Even Mix"), m=N/X novices per expert, each expert's productivity falls from 1 to something like 1/(m+1). The group's total productivity is now

$$(X*X/ (N+1)) + N*n$$

which is shown in the lower curve in the figure. The figure shows the productivity of DAY CARE versus the even mix. This graph shows the total productivity for the team in units of experienced people's productivity. As the number of novices increases, the even mix line shows the effect of

training them. Let us check that the assumed productivity difference is not skewing the results. The lower figure shows the ratio of DAY CARE to even mix, for different productivity assumptions. Note that with five experts and five novices, the ratio is actually just below one, meaning that the experts are absorbing and making use of the novices. By two novices per expert, DAY CARE is already considerably more effective.





The nature of the training does not matter. Design and teaching are antagonistic tasks [as described in TEAM PER TASK (4.1.21)] that are better split into separate teams.

Treating the delivery of trained people as separate from the delivery of running software gives you access to OWNER PER DELIVERABLE (A.5.19). SOMEONE ALWAYS MAKES PROGRESS (4.1.20) protects the delivery of running software.

Sample situations:

A. Mentoring.

The standard recommendation in the industry is to assign one to five novices to each trained expert. The consequence is that the experts spend the prime part of their energies training halfheartedly. Besides being drained of the energy needed to design the system, the experts typically do not have the personality, background, or inclination to actually teach the novices how to do design. They are torn between trying to get maximum productivity out of their trainees and trying to perform the maximum amount of development themselves. Thus, they neither develop the system nor train the novices adequately.

Some companies have developed dedicated "Apprenticeship" programs, in which novices are put under the tutelage of a dedicated mentor for 2 out of every 3 weeks for 6 months.

B. Adding staff.

In *The Mythical Man-Month*, Fred Brooks talks about the training costs of adding people to a project [Brooks 1995]. These new people drain productivity from the experts. The same suggestion applies: Put the newcomers in a separate team to learn the system and move them to the progress team as soon as they are up to speed.

In *Situated Learning: Legitimate Peripheral Participation*, Lave and Wenger describe the use of this sort of arrangement in apprentice-based work situations [Lave Wenger 1991].

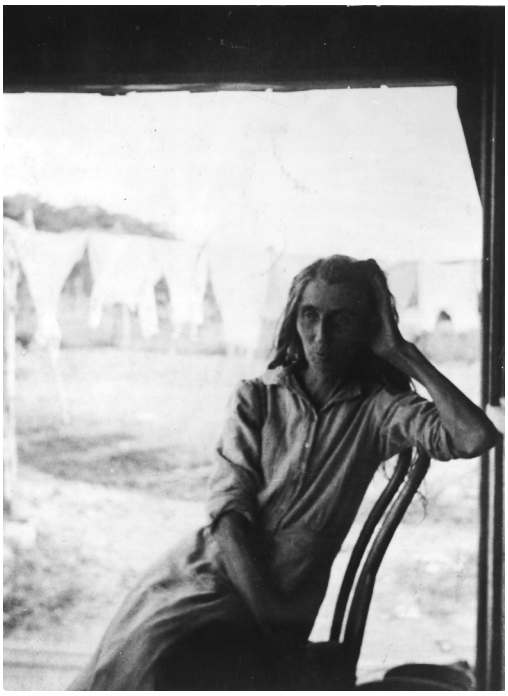
A version of this pattern first appeared in [Cockburn 1998].

Related patterns:

This pattern is a cross-specialization of several patterns: Someone Always Makes Progress (4.1.20), Team Per Task (4.1.21), Sacrifice One Person (4.1.22), and Owner Per Deliverable (A.5.19).

See also Apprenticeship (4.2.4).

4.1.24 Mercenary Analyst *



On one of his many journeys in the Appalachian Mountains, the itinerant folk song collector John Jacob Niles heard a woman singing a particularly beautiful song. He persuaded her to repeat the now-famous Christmas song "I Wonder as I Wander" until he had learned it himself. He later said, "I never saw her again."

... 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.) Supporting both a design notation and the related project documentation is too tedious a job for people who are directly contributing to product artifacts.

✣ ✣ ✣

Technical documentation is the dirty work required of every project. It's important to create—and, more so, to maintain—good documentation for subsequent use by the project team. But who writes these documents?

If developers create their own documentation, "real" work is hampered. Meeting software deadlines means money to the organization, and technical documentation is one of those things we tell ourselves that we can defer until there is time to do it. But the time often never comes,

and an organization without good internal technical documentation of its system has a serious handicap.

Internal documentation is often write-only: it is rarely read after it is written.

Engineers often don't have good communication skills.

Many projects use tools like Rational Rose to do design. These tools produce pretty pictures. A good picture, however, is not necessarily a good design, and architects can become victims of the elegance of their own drawings (see the following rationale).

Therefore:

Hire a technical writer who is proficient in the necessary domains, but who does not have 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 use by the organization itself.

The documentation itself should be maintained online if possible. It must be kept up to date (therefore, MERCENARY ANALYST is a full-time job), and it should relate to customer scenarios [SCENARIOS DEFINE PROBLEM (4.2.8)]. Note, though, that all team members need to provide input to keep the documentation current. The AD-HOC CORRECTIONS (A.5.2) pattern [Weir 1998] suggests that a master copy of the documentation be kept and that team members write corrections in the margin. One team member is assigned to periodically update the document.

The success of this pattern depends on the ability to find 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 [STAND-UP MEETING (5.2.7)] and monitored by community experts outside the project.

If the MERCENARY ANALYST really is a "mercenary" who, as Paul Chisholm notes, "rides into town, gets the early stuff documented, kisses his horse, saddles up his girl, and rides off into the sunset" [Chisholm 1994], then it's good to retain some of the expertise by combining MERCENARY ANALYST with DEVELOPING IN PAIRS (4.2.28).

This pattern, uncommon though empirically grounded and effective, is found in Borland's Quattro Pro for Windows and in many AT&T projects (e.g., a joint venture based in New Jersey, a formative organization in switching support, and others). It is difficult to find people with the necessary skills to fill this role.

Rybczynski writes:

> 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. [Rybczynski 1989, p. 121].

A passage from Manzoni's *The Betrothed* [Manzoni 1984] might amuse the MERCENARY ANALYST.

> The peasant who knows not how to write, and who needs to write, applies to one who knows that art, choosing as far as he can one of his own station, for with others he is hesitant, or a little untrusting. He informs him, with more or less clarity and orderliness, of who his ancestors were, and in the same manner tells him what to set down on paper. The literate person understands part and guesses at the rest, gives a few pieces of advice, suggests a few changes, and says "Leave it to me."
>
> He picks up his pen, puts the other's thoughts as well as he can in literary form, corrects them, improves them, embellishes them, tones them down, or even omits them, according to how he thinks best, because—and there's nothing to be done about it—someone who knows better than others has no wish to be a mere tool in their hands, and when he is concerned with the business of others he wants it to go a little in his own way.

Richard Gabriel [Gabriel 1995] notes the following are important traits of this role:

- Possesses strong skills as a meeting facilitator
- Likes things organized
- Possesses good attention to details
- Possesses written instructional material (for software)
- Lacks the ego to invest in the material being documented
- Is very smart and highly educated

In exceptional cases, the MERCENARY ANALYST can actually have a stake in the design. Betsy Hanes Perry writes:

> When I fill this role, I most definitely have a stake in the design: I want to make sure it's elegant, consistent, and clean. The architect has primary responsibility, of course, but I also suggest places in which the design conflicts with itself or may lead to future misunderstandings. As I see it, a software architecture is an idea. The designer/implementors are responsible for expressing that idea (or those ideas) as code; I express it/them as prose. Both are projections of the idea into a particular plane. When there's a conflict, the code is probably correct [Perry 1997].

Many projects put faith in tools and notations such as Unified Modeling Language (UML) to improve quality. But, as Perry points out, tools largely provide the forum and opportunity for a human being to engage in the processes and convey the insights that contribute to quality. For documentation to have added value as a quality tool, the documentation process must proceed in the spirit of this admonition.

Paul Chisholm offers the following about the history and rationale of MERCENARY ANALYST:

> MERCENARY ANALYST came from two sources:
>
> (1) Borland's Quattro Pro for Windows, which Jim Coplien identified as *the* most productive software development organization he's ever seen (average 1000 delivered noncommentary source lines of C++ per staff **week**), in large

part due to the fact that developers had people to write the development documentation for them).

Designer/coders have responsibilities that cannot be delegated. Some responsibilities, such as documentation, can be delegated. Besides, many excellent programmers and most average ones are less than stellar writers. (Richard [Gabriel] may disagree that this *is* the case, and will certainly disagree that this *should* be the case ...)

(2) A combination of two patterns. One, from Tony Hansen's group, is DISPOSABLE ANALYSIS: do analysis once, translate to design, throw away the analysis, keep only the design up to date with the code. The other is my observation that most CASE tools require significant experience in the method and the tool itself. If you have DISPOSABLE ANALYSIS (which few projects plan to do but many follow unintentionally), you should not develop local expertise in Computer-Aided Software Engineering (CASE) tool operation.

It's bad enough learning FrameMaker. CASE tools tend to have lousy user interfaces; it's a real pain to use them, or learn how to use them.

The "mercenary" in MERCENARY ANALYST comes from the "hired gun" quality a MERCENARY ANALYST might have: he rides into town, gets the early stuff documented, kisses his horse, saddles up his girl, and rides off into the sunset. That's the DISPOSABLE ANALYSIS model, not the Borland Quattro Pro for Windows model!

MERCENARY ANALYST plays well with DEVELOPING IN PAIRS (4.2.28).

Someone quoted by Jim Coplien wrote that "Mercenary Analyst is the professional technical writer who takes care of all the project diagrams and documentation so that the task of documentation doesn't get in the way of the architects."

Maybe not a "tech writer," and not "*all* the diagrams and documentation," but, yes, that's the idea.

What should be a MERCENARY ANALYST's education? Mastery of his or her tools (e.g., word processor, CASE tool) is beyond that of most users. Experience (perhaps expertise) in the "method" behind the documentation (e.g., an ObjecTime MERCENARY ANALYST would have to know ROOM well, someone writing requirements would need systems engineering and/or software development experience).

What is the MERCENARY ANALYST's motivation? To get the *software* (**not** the documentation) out faster!

How can one paint CASE diagrams without knowledge of software? I had some naive hope that a CASE tool MERCENARY ANALYST could be a highly skilled clerk. I've given up on that. There may be some way of combining MERCENARY ANALYST with DEVELOPING IN PAIRS (4.2.28) (or a variant for triples) to make MERCENARY ANALYST some sort of entry-level or apprentice position.

Domain Knowledge. While knowledge of the domain is important for a project [(DOMAIN EXPERTISE IN ROLES (4.2.22)]. I don't think the MERCENARY ANALYST needs it. (I hope not!)

Knowledge of software is important. Would you trust a driving instruction manual written by someone who'd never driven? [Chisholm 1994]

4.1.25 Interrupts Unjam Blocking **

During one project status meeting, it was reported that a critical piece of hardware was malfunctioning. Unfortunately, the expert on the hardware was on the other side of the country and was involved in his own work. However, the expert had the (mis)fortune to be on the meeting conference call, as was the project director, who informed the expert in blunt terms that his services were required immediately. He was on the next plane out.

... you are fine-tuning the scheduling for a high productivity design/implementation process or a low-latency service process. The scheduling problem is to be addressed on a small scale (i.e., this is not the scheduling for entire departments, but the work of cooperating individuals). You want to use INFORMAL LABOR PLAN (4.1.14), but you need additional criteria for individuals and small groups in order to plan their schedules. Local decisions may lack the scope necessary to avoid duplication of work, missed opportunities, and other unfortunate problems.

A comprehensive scheduling plan is difficult if not impossible to develop; yet, without some kind of plan it becomes easy to fall into thrashing: rushing from task to task without making any real progress.

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

Complete scheduling insight is impossible. Even if it were possible to capture the entire picture of the project for an instant, that picture would change very quickly. The dynamics of project development mean that the best we can hope for is a high-level, approximate schedule.

The programmers with the longest development schedules will benefit if more of other peoples' code is done before they try integrating or testing later code, especially if their interval can't otherwise be shortened [see CODE OWNERSHIP (5.2.13)].

Therefore:

If a process is about to be blocked because a critical resource is unavailable, interrupt the role that provides that resource so that role can unblock that process.

The nature of the critical resource can vary. It may be a software module that is in the critical path. It could be the latest software integration. It is often critical knowledge, without which one cannot move forward. Whatever the resource, the approach is to interrupt the provider of that resource.

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

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 WORK FLOWS INWARD (4.1.18), RESPONSIBILITIES ENGAGE (5.1.14), HALLWAY CHATTER (5.1.15), MOVE RESPONSIBILITIES (5.1.18), and COUPLING DECREASES LATENCY (5.1.22).

This pattern is intended to apply most frequently to cooperating developers working on a single project, a concept supported empirically from a high productivity process in AT&T. There are strong software engineering (operating system) principles at work here as well.

It may be useful to prioritize interrupts and to service the ones that would optimize the productivity of the organization as a whole. That is, it is better to unblock four people who are currently blocked than to unblock a single squeaky wheel. The decision-making process should be fast, and most of the time it should be distributed. Where arbitration is needed, apply PATRON ROLE (4.2.15). The simplest resolution is the pattern DON'T INTERRUPT AN INTERRUPT (4.1.26).

The PATRON ROLE (4.2.15) and manager roles can help the team audit the project for blocked progress, but they should defer to the Developers (or other directly impacted roles) to resolve the blockage whenever possible. Management intervention can be effective, but it may risk goodwill within the project.

Joe Maranzano notes that a corollary to this pattern is another pattern, which can be summarized as follows: Don't put too many critical tasks on one person [Maranzano 1992]. This pattern is related to MODERATE TRUCK NUMBER (4.2.24) and DISTRIBUTE WORK EVENLY (5.1.13).

This pattern is much less effective if the provider of the resource is not working on the same project as you are. In that case, the provider has little incentive to service your interrupt, and you risk alienating the provider if you engage in incessant pestering. This problem can be mitigated by adopting a policy of reciprocity, of fair and proactive exchange of value among partners. [Dikel 2001].

4.1.26 Don't Interrupt an Interrupt *

The original interruption device.

... you've applied INTERRUPTS UNJAM BLOCKING (4.1.25), but you notice that the organization is now thrashing, particularly in the end game or under heavy churn.

It's important to balance a desire that SOMEONE ALWAYS MAKES PROGRESS (4.1.20) with the thrashing that can accompany short-term priority calls. One worker will inevitably be blocked on you—you can't do both things at once. Complete foresight and perfect scheduling are unreasonable to expect.

Therefore:

If a developer is already working in "interrupt mode" on a critical issue, don't put that work aside until it is complete or until that issue itself becomes hopelessly tangled.

This pattern prevents the endless churn that can result from too much context switching. It also helps to ensure that SOMEONE ALWAYS MAKES PROGRESS (4.1.20). And it provides some "back pressure" in the process that can help temper irresponsibly quick reversals of position in the front end.

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

This concept relates to the "red zone" from Linda McLyman's analysis of the Satir change model [Satir 1991], which suggests that if a foreign element (problem) arrives before the organization starts to learn its way out of the last foreign element, recovery is difficult.

4.2 Piecemeal Growth Pattern Language

The Pattern Language

This pattern language offers patterns to strengthen and fine-tune an organization using feedback and insight. It is essentially a process of repair. Figure 4.2 shows the patterns and their connections to each other.

Note, perhaps surprisingly, that none of these patterns have fundamental ties to software development. They are applicable to any design activity that involves a group of people building something to solve a problem. They are as applicable to software services as they are to building product, to hardware development as they are to software development. They are patterns about human nature and human organizations, about the ways that people come together to solve problems.

A Story about Piecemeal Growth

When I started to plan the Q project, I wanted a small core team of architects, so I employed SIZE THE ORGANIZATION (4.2.2) with an eye to PHASING IT IN (4.2.3) through APPRENTICESHIP (4.2.4) with other staff later on. The project was too large for a SOLO VIRTUOSO (4.2.5) approach—though we would use that pattern later to flesh out a prototype. I put forward the opportunity and made it possible for people to sign up; there was no corporate or management compunction to join. Hence, it was purely a SELF-SELECTING TEAM (4.2.11), started as a SKUNKWORKS (4.2.14) beneath management radar.

My main job as project coordinator was to put up the FIREWALLS (4.2.9) to management until we had our act together. But my second job was to make sure we got a good group of people to the end of HOLISTIC DIVERSITY (4.2.19). We brought in Lalita for her work in scripting languages and their environments and Peter for his architectural expertise. Later, we decided we needed market domain knowledge, and that's when we brought on Jim and Beki in the interest of having DOMAIN EXPERTISE IN ROLES (4.2.22). The recruitment strategy always involved ferreting out matches of interest that would excite the players, amplified by the new nature and somewhat subversive approach of the opportunity. TEAM PRIDE (4.2.13) was an emergent property of this process. We also had our own value system and model of rewards: all team members would share credit for any patents that were issued, and, together, we would seize a leadership role in the organization. We also knew that we were catering to the organization's product interests and that the organization would COMPENSATE SUCCESS (4.2.25).

Beki served as the GATEKEEPER (4.2.10), bringing in ideas from the AOL Instant Messenger world, interviewing (child!) users of the system, and bringing in knowledge of the organization and market opportunities. She and I split the duties of PUBLIC CHARACTER (4.2.17) and MATRON ROLE (4.2.18).

We moved forward on design using CRC cards to formulate an architecture, employing SCENARIOS DEFINE PROBLEM (4.2.8) and GROUP VALIDATION (4.2.32). The goal was to get the project "running" on CRC cards and then to implement a first, simple cut in a 1- or 2-day programming session, DEVELOPING IN PAIRS (4.2.28) all together in one room. The CRC cards were given to the individuals best suited to those areas, exemplifying both DOMAIN EXPERTISE IN ROLES (4.2.22) and, to the degree one could talk about subsystems at that point, SUBSYSTEM BY SKILL (4.2.23).

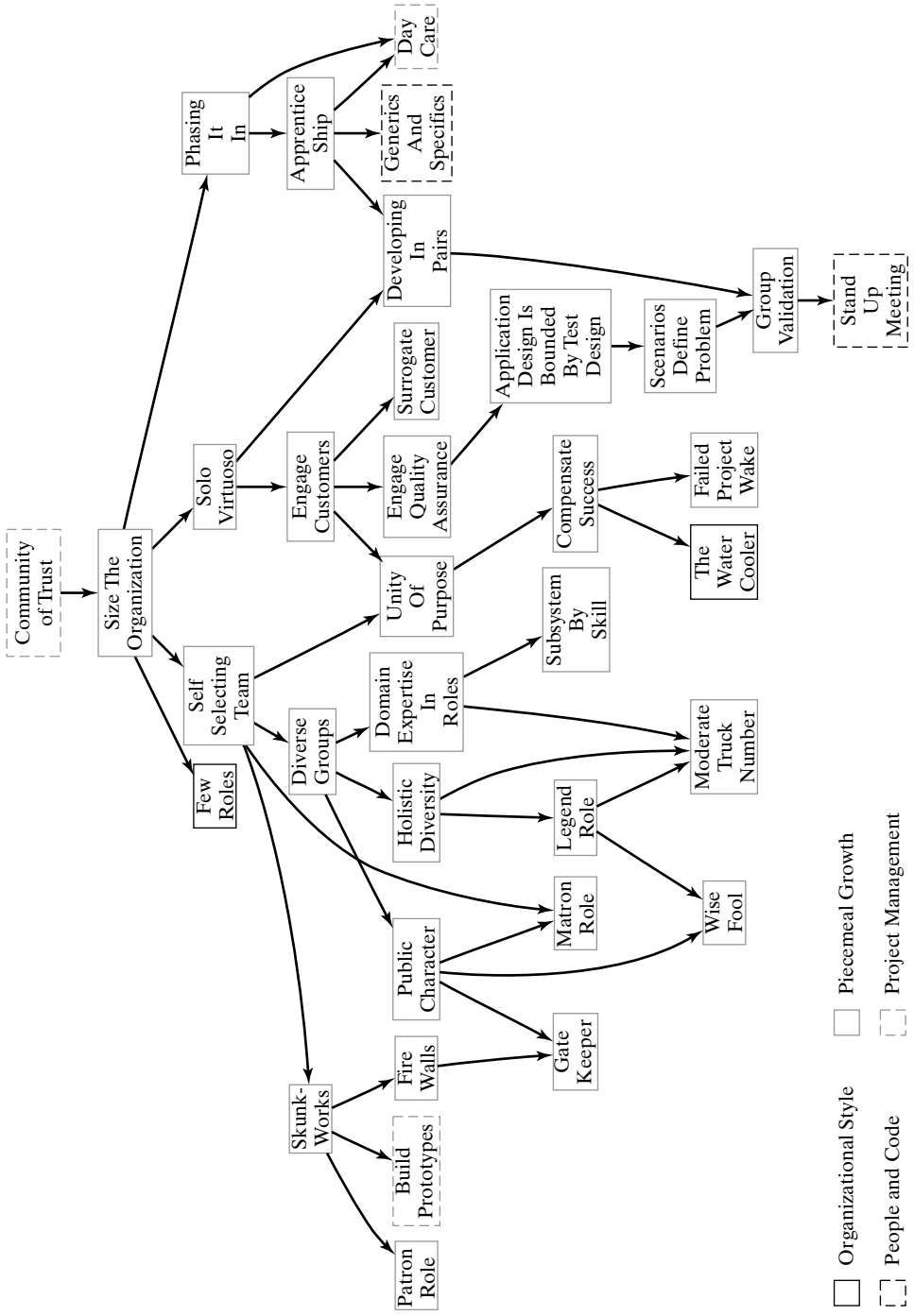


FIGURE 4.2 The patterns and their Interconnection

At our (frequent) meetings, we made sure that work was spread around evenly. We did most things in a group to make sure that the specialization didn't get out of hand. We occasionally traded off CRC cards, all in the interest of having a MODERATE TRUCK NUMBER (4.2.24).

At some point in the process, people felt that the CRC cards weren't enough and that we needed to document the scenarios. We used ping-pong diagrams, first on whiteboards and later using a formal documentation tool [SCENARIOS DEFINE PROBLEM (4.2.8)]. In order to create this documentation, we had to SACRIFICE ONE PERSON (4.1.22) and find a MERCENARY ANALYST (4.1.24) [we were too small to enlist a full-fledged MERCENARY ANALYST (4.1.24), but we faked it].

Lalita went away as a SOLO VIRTUOSO (4.2.5) to BUILD PROTOTYPES (4.1.7). The prototype failed to energize the team to take the next steps forward, and things came to an impasse, particularly in light of competing priorities on other development projects.

Dysfunction struck the organization due to the untimely departure of Beki and Peter from the project and, afterwards, to Lalita's promotion out of the project. Jim took the ideas forward into another project, but he took no other people with him. We did not have a FAILED PROJECT WAKE (4.2.26), though perhaps we should have. We didn't get so far as to run the development exercise as a team in a room, at which point INTERRUPTS UNJAM BLOCKING (4.1.25) and DON'T INTERRUPT AN INTERRUPT (4.1.26) would have become important.

4.2.1 Community of Trust

See Section 4.1.1.

4.2.2 Size the Organization **

... within a larger organization, usually that of a sponsoring enterprise or company, there need to be smaller organizations capable of creating large software systems [greater than 25,000 source lines of code (SLOC)] that meet competitive cost and schedule benchmarks. This pattern shows how the proper sizing of an organization is vital to the health of the project and to the productivity of its people.

Large software projects (greater than 25,000 SLOC) are seldom delivered on time and within budget when the development team is either too large or too small.

Two arguments have led us to this conclusion:

1. There are limits to the size of software development teams that allow them to work effectively. A team can handle a larger problem than an individual can ([Beyer Holtzblatt 1998], p. 4).

2. The addition of people to a project late in the game rarely helps complete that project on time and within budget.

If a software development team is too large, you can reach a point of greatly diminishing returns. We have found empirically that an organization's size affects a deliverable nonlinearly. Communication overhead goes up as the square of the size increases, which means that the organization becomes less cohesive as the square of the size increases, while the "horsepower" of the organization goes up only linearly.

In addition, if the organization is too small, the team won't have critical mass, and productivity will suffer. Projects larger than 25 KSLOC can rarely be done by a SOLO VIRTUOSO (4.2.5), and overly small organizations have inadequate inertia and can easily become unstable.

However, experience has shown that a carefully selected and well-nurtured small team of around 10 people can provide a suitable critical mass with a capacity to develop a 1,500 KSLOC project in 31 months, a 200 KSLOC project in 15 months, or a 60 KSLOC project in 8 months.

Keeping the organization small makes it possible for everybody to have knowledge of how the project works (*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).

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 ensure both buy-in and benefit can occur only in small organizations. Tom DeMarco has noted that everybody who is to benefit from process should be involved in process work and process decision making [DeMarco 1993].

Further study might evaluate the relationship between this pattern and Alexander's THE DISTRIBUTION OF TOWNS ([Alexander 1977], ff. 16) and related patterns. Here, we stipulate that the social organization must be small reflecting a SUBCULTURE BOUNDARY ([Alexander 1977], ff. 75) and an IDENTIFIABLE NEIGHBORHOOD ([Alexander 1977], ff. 80). Alexander emphasizes the grander architectural context that balances support for the ecology with the economies of scale that large towns can provide, while also supporting the xenophobic tendencies of human nature. Small organizations like that being built here rarely exist in isolation, but instead exist in the context of a broader supporting organization. This relationship to the larger organization invokes the PATRON ROLE (4.2.15).

Adding people late to a project rarely helps complete that project on time and within budget.

One manager writes: "On [one] project, I grew from 10 to 20 people to meet a customer contract ... with new people, [I] wound up three months late because of 'absorption' of new folks into the organization."

Many software development cultures support technical manager groups of up to 10 people. Adding more people to the group would force a group split, which can cause a large decrease in productivity, all other things being equal. We have also found that a single team is better than a collection of subteams. The faster a team breaks up into subteams that worry about their own responsibilities rather than those of the larger team, the less effective the enterprise will be as a whole.

Therefore:

By default, choose about 10 people to establish critical mass in the development of large software systems and avoid adding individuals late in the game or trying to work backwards from a completion date.

Experts vary on the exact number. The number 10 has a bit of tradition associated with it, but numbers like 6 or 7 are also common. A two-person group is too small, and a 13-person group is too big.

Starting a large project with 10 people can be overkill, but it avoids the expense and overhead of adding more people later. However, once a core team establishes an identity, it can grow graciously by PHASING IT IN (4.2.3) or using APPRENTICESHIP (4.2.4). The organization can generate knowledge early on by building and throwing away a prototype [see BUILD PROTOTYPES (4.1.7)]. To decide whom to hire into the nascent organization, use patterns like DOMAIN EXPERTISE IN ROLES (4.2.22) and ARCHITECTURE TEAM (5.2.4). SMALL WRITING TEAM (A.5.27) ([Bramble 2002], p. 31) suggests that two or three people write the use cases; the others will fill in other roles.

Astute readers might consider this pattern and remark, "You have a strange idea of what constitutes a large project! I can see this pattern working for projects that will grow to 30 or 40 people and maybe a few tens of thousands of lines of code. But does this pattern work for really large projects?"

First, it's important to understand that few real software development teams are larger than a few dozen people; larger projects almost always self-organize into subcommunities [DIVIDE AND CONQUER (5.1.6)]. But even the largest projects start with an idea, and an idea starts with an individual or a small group of people. This pattern says that a small group should take the project as far as they can before other staff are actively engaged. One must anticipate the point of diminishing returns for the seed team and seek people early enough so that they will be available and ready when they are needed. And, of course, people should be brought on gradually (see PHASING IT IN (4.2.3), DAY CARE (4.1.23), etc.). But start small, and stay as small as possible for as long as possible. Large systems grow from small systems that work.

Second, remember that it is imperative to have FEW ROLES (5.1.2). With 10 people, it is easy to define and fill half a dozen or so roles. But with a large initial team, people will at first be at a loss as to what to do until they receive assignments (and you can't give everyone an assignment all at once). So they will find something to do, and they will tend to invent roles for themselves. Unfortunately, this is a good way to create deadbeat roles.

Joe Walters said that a project shouldn't grow larger than the size of the auditorium of the building where the project is centered [Walters 1976].

Once the staff sizing is complete, the project can SIZE THE SCHEDULE (4.1.2).

4.2.3 Phasing It In **

... key project players have been hired or otherwise brought into the project. They cover the necessary areas of expertise [DOMAIN EXPERTISE IN ROLES (4.2.22)], but the project still needs more staff.

Growing projects must figure out how to increase the number of long-term staff: whom to hire, how many to hire, and when to hire them. Projects must ramp up while minimizing the pains of growth.

You need enough people for critical mass. Yet you cannot just hire anyone off the street; staff are not plug compatible and interchangeable.

The right group of initial people [SIZE THE ORGANIZATION (4.2.2)] sets the tone for the project, and it's important to hire the key people first. You need a critical mass of key people early on. However, too many people too early creates a burden for the core team.

Therefore:

Phase the hiring program. Start by hiring people to meet the basic core competencies of the business and gradually bring on new people as the project needs to grow.

The organization can staff up to meet development load. This pattern is closely related to APPRENTICESHIP (4.2.4) and MODERATE TRUCK NUMBER (4.2.24). DAY CARE (4.1.23) can be applied to help with the training and mentoring load that new employees place on the organization.

This well-known management technique allows the project to establish an identity early on and to grow graciously.

Larry Putnam points out that projects that grow very quickly at the beginning tend to be late. He advocates the idea of growing staff gradually [Putnam 1992].

In *The Mythical Man-Month,* Brooks states, "V. A. Vyssotsky of Bell Telephone Laboratories estimates that a large project can sustain a manpower buildup of 30 percent per year. More than that strains and even inhibits the evolution of the essential informal structure and its communication pathways" ([Brooks 1995], page 293).

What constitutes *core competencies*? It depends on the business you are in. If you are in finance, you want people who can develop financial software. The better people you can get early on, the better off you will be, and it is probably a good return on investment to spare no expense on talent at this early stage. Talent isn't limited to domain knowledge, though. You also need individuals who can put customers at ease, keep a cool head for strategic planning, "fill in the cracks" by performing the miscellaneous detailed tasks that others don't want to do or forget to do, etc. Many individuals have many of these talents. The key, then, is to cover the crucial needs early on with as few people as possible and to grow the organization once that organization has gelled [see STABLE ROLES (5.1.5)]. You can achieve these goals with HOLISTIC DIVERSITY (4.2.19) and DIVERSE GROUPS (4.2.16).

4.2.4 Apprenticeship *

...the project is incrementally staffing up after the first round of experts have been brought on board.

✣ ✣ ✣

A project must balance its need for growth with its need to develop and maintain deep domain expertise. You need enough people for critical mass. However, staff are not plug compatible and interchangeable. And academic training and prior experience are rarely, in themselves, adequate preparation for competent work at a new task.

Therefore:

Turn new hires into experts [see DOMAIN EXPERTISE IN ROLES (4.2.22)] through an apprenticeship program. Every new employee should work as an apprentice (not just a mentee) to an established expert. Most apprenticeship programs will last 6 months to 1 year—the amount of time it takes to make a paradigm shift.

✣ ✣ ✣

It will be possible to maintain expertise in the organization. This pattern also reduces the organization's *truck number* by spreading knowledge around. The truck number is the number of

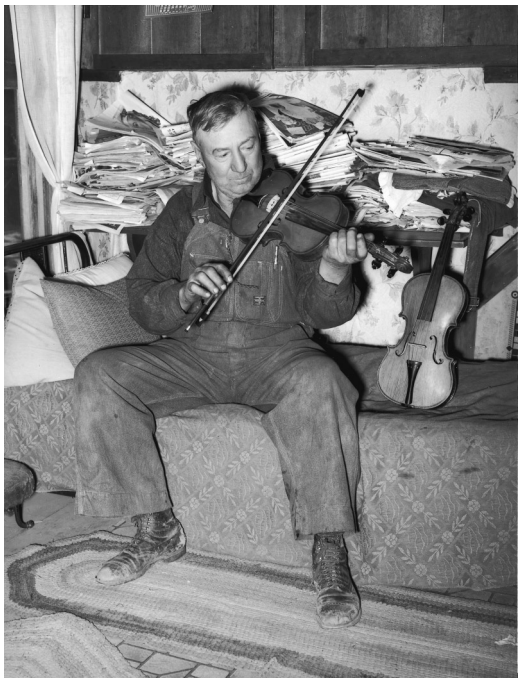
people who possess unique critical domain expertise. If any one of these people were hit by a truck, the organization will have lost a critical resource [see MODERATE TRUCK NUMBER (4.2.24)]. By working to reduce the truck number, the experts feel valued, and the apprentices are given a good environment in which to learn.

Manage drain on expert staff resources with DAY CARE (4.1.23).

DEVELOPING IN PAIRS (4.2.28) is often used as an effective APPRENTICESHIP technique.

It is better to apprentice people than to put people through a "trial by fire" that may damage the project. The apprenticeship approach makes it possible to form domain-specific teams, and it is important to keep the team concept as a central part of organizational values.

4.2.5 Solo Virtuoso *



. . . we have described the optimal size of organizations needed to create large software systems on time and within budget [SIZE THE ORGANIZATION (4.2.2)]. The following pattern explains what to do for smaller systems (fewer than 25 KSLOC) when a product must still be created on time and within budget, but when rapid growth is not anticipated after the first release.

When a smaller software project (fewer than 25 KSLOC) is overstaffed, communication overhead increases, and talented individuals, who could produce the software entirely on their own, are bridled, their "horsepower" diminished.

We have said that organizational size affects the deliverable in a nonlinear manner [SIZE THE ORGANIZATION (4.2.2)]. We have also observed that communication overhead increases 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 increases only linearly.

The question, then, is this: What organizational size works best for smaller software projects?

The answer depends on the individual(s) involved in the project. The productivity of a single individual can be higher than that of a collection of productive individuals. We have seen single-person developments generate 25 KSLOC of deliverable code in 4 months (e.g., for a

craft interface for a telecommunication system) and two-person developments generate 135 KSLOC in 30 months. Many of these individuals adhered faithfully to all stipulated review schedules and verification steps.

Boehm [Boehm 1981] 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."

[Note: Boehm quotes Grant and Sackman ([Grant Sackman 1966] p. 667), who note a 26-fold spread.]

The result of using a SOLO VIRTUOSO is an organization limited to producing small development efforts. Though there is a single development role, other roles may be necessary to support marketing, toolsmithing, and other functions. The productivity of a suitably chosen single developer is sufficient for sizable projects; here, we establish 25 KSLOC as a limit.

Therefore:

Design and implement the entire system with one or two of your most effective developers.

This pattern is not a "License to Hack." The work of a SOLO VIRTUOSO is still subject to technical reviews, validation, and verification at appropriate times in the development cycle [ENGAGE CUSTOMERS (4.2.6) and STAND-UP MEETING (5.2.7)]. This pattern works nicely with DEVELOPING IN PAIRS (4.2.28).

See also MODERATE TRUCK NUMBER (4.2.24), which raises concerns about the use of this pattern in risk-averse businesses.

4.2.6 Engage Customers **

Clerk measuring a customer for a suit of clothes, San Antonio, Texas.

A friend of one of the authors once designed and implemented the user interface for a large system. He received input from customers on how to make it useful for them. Unfortunately, the requirements writers had a different idea and made him remove the features the customers liked. However, when the customers asked for the missing features, the requirements writers were forced to relent. I guess the situation didn't help relations between my friend and the requirements writers.

... an organization is in place, and its QA function has been generally shaped and chartered. The QA function needs input to drive its work. Many people in the enterprise are concerned about quality issues.

✣ ✣ ✣

It's important that the development organization ensures and maintains customer satisfaction by encouraging communication between customers and key development organization roles. This communication isn't the responsibility of any single "customer satisfaction" group; instead, the need pervades the entire organizational structure. Most organizations hesitate to allow direct contact between developers and customers, fearing that the developers are "loose cannons on deck" who will promise to deliver things that exceed the scope of a job.

Yet, you can't know all of the requirements up front, so developers need to keep going back to customers for more information. Customers, in turn, need to keep coming back to developers with their insights, particularly when developers BUILD PROTOTYPES (4.1.7). After all, requirements changes occur even after design reviews are complete and coding has started.

Many organizations depend on their marketing organization to provide them with requirements data. But marketing doesn't provide design data (Beyer Holtzblatt 1998], p. 30). The best that marketing can do (or *should* do) is to understand what will sell and why people will buy what you want to sell. Designers, in turn, must understand how people will use the product in a way that creates value for them. Good value sometimes leads to good market potential, but marketing usually looks at other factors (e.g., brand name recognition, product name, and posturing in the market) that designers care little about.

Missing customer requirements are a serious problem; in fact, most problems in software systems can be traced to requirements problems ([Daley 1977], [Boehm 1976]). Yet it seems like so much effort to elicit these requirements, especially when this work does not directly produce a marketable artifact. It seems like make-work.

Customers are traditionally not part of the mainstream development effort, which makes it difficult to discover and incorporate their insights. Yet, customer contact correlates with project success [Keil Carmel 1995].

Trust relationships between managers and coders are often strained, so you don't want them to be the sole intermediaries between developers and customers.

Therefore:

Closely couple the Customer role with the Developer and Architect roles, not just with QA or marketing roles. In short, developers and architects must talk freely and often with customers. When possible, engage customers in their own environment rather than bringing them into your environment.

Two things are necessary to allow this interaction to happen: opportunity and culture. Developers must have the opportunity (and the means) to communicate with customers. They should meet customers personally to establish trust and a free flow of communication.

But these visits will be superficial if the organizational culture builds walls between customers and developers. In particular, if system requirements must go through a lengthy formal process to be approved, the developer will be hamstrung, unable to respond to customer requests. Therefore, the organization must develop a culture where developers have some latitude to respond to customers. We are not saying, however, that all control of requirements should be relegated to the developer. Order is necessary.

Beyer and Holtzblatt note that "many common ways of working with customers remove [designers] from their work" ([Beyer Holtzblatt 1998], pp. 36–7). One way to solve this problem is by "putting designers and engineers directly in the customer's work context" ([Beyer Holtzblatt 1998], p. 20), which is particularly important if you are using customer engagement to create wholly new market directions for the enterprise, rather than simply refining existing work. Putting developers in the customer's work environment also trains developers' intuition about good design and good human interfaces, and this intuition can fill in when specific detailed requirements are unavailable ([Beyer Holtzblatt 1998], p. 35).

Language is a key element of culture that can ensure a smooth customer engagement if treated properly and smother it if treated badly. Don't make your customers learn UML or other technical notations; instead, do your best to learn *their* language and to communicate with them in the terms of *their* culture.

The QA team can monitor the relationship to keep the direction within contractual business limits, while allowing a free flow of insights back and forth between developers and customers. Such communication can often flow unimpeded; at other times, however, it cannot [see GATEKEEPER (4.2.10)].

Note that this pattern is all about relationships and culture. It is the culture of respect for and interaction with customers that makes the communication effective, such as during the writing of use cases, as described in PARTICIPATING AUDIENCE (A.5.20) ([Bramble 2002], p. 35).

This pattern supports requirements discovery from the customer, as required by SCENARIOS DEFINE PROBLEM (4.2.8) and BUILD PROTOTYPES (4.1.7). Other patterns like FIREWALLS (4.2.9) also build on this pattern. The pattern RECOMMITMENT MEETING (4.1.12) is a more formal derivative of this pattern in a different context.

A good understanding of customer needs can help you to avoid rework after implementation is done. While it is also important to continuously engage customers through each development episode of iteration, early understanding helps launch the effort in the right direction. For example, a Navision team in Copenhagen felt that improvements in customer engagement helped save time on their development schedule [Pedersen 2002].

This pattern was prominent in the Borland Quattro Pro for Windows case study. Also, see [Floyd 1992] and, in particular, the works of Reisin and Floyd therein.

Some processes and methods are founded on customer engagement, such as IBM's Joint Application Development (JAD). Other methods are conducive to customer engagement, such as Cunningham and Beck's CRC design technique. Other methods, and especially most CASE-based methods, are indifferent or harmful to customer engagement.

Even some of the best customer engagement techniques tend to stop once they achieve some level of contractual agreement about what is to be delivered. Customer engagement in agile processes, however, goes far beyond this traditional stopping point. Developers need to assimilate the context in which their product will be used: This process is called *contextual design.* Contextual design means gathering data on customers' models of how they do their work rather than creating models of how the program will solve the problem (such as done in use case modeling). See [Beyer Holtzblatt 1998].

The pattern is called ENGAGE CUSTOMERS (in the plural) to support a domain view and to avoid the possibility of being blindsided by a single customer.

The project must be careful to temper interactions between Customers and Developers, using FIREWALLS (4.2.9), GATEKEEPER (4.2.10), and the QA organizational presence as in ENGAGE QUALITY ASSURANCE (4.2.29). A big part of interacting with customers involves learning how they want to interact with the project as the unfolding software uncovers problems in requirements and systems engineering [see APPLICATION DESIGN IS BOUNDED BY TEST DESIGN (4.2.30)].

Note that maintenance of 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 percent of the time) or when the attention they receive is rude or unhelpful (50 percent of the time). If customers experience software problems that cost more than $100 to fix and if the company does not fix the problems,

then only 9 percent of the customers would do business with the company again. 82 percent would do business with the company again if the problems were quickly resolved after they complained. (The source for the former pair of percentages is The Forum Corporation; the source for the latter pair of percentages is Traveler's Insurance Company [Zuckerman Hatala1992].)

Joe Maranzano [Maranzano 1992] notes that this pattern probably should come earlier in the pattern language. However, it is important that the project roles be defined first—particularly those that involve interacting with the customer, and those that are driven by customer input (e.g., QA). Said in another way, the organization exists to serve the customer, so the organization should be in place before the customer is fully engaged.

This pattern works only if customers are directly accessible to the development team. If such accessibility is impossible for business reasons or because of geographic separation, consider SURROGATE CUSTOMER (4.2.7).

4.2.7 Surrogate Customer *

Store dummy displaying Daniel Boone hat, Amsterdam, New York.

... the project is beginning to move forward. As architects and developers get deeper into the project, questions about requirements begin to surface.

It is important to exchange ideas and clarify issues with customers. However, customers may not be available.

There are several reasons that a customer may be unavailable. If the project is new, there may be no customers yet. In fact, the product might even create its own customers. Or, the organization may never have established relationships with existing customers, and now it may not be a propitious time to do so.

In some cases, customers might not have the time to meet right now. They're busy too. But you need answers immediately.

Some corporate cultures insulate the developers from the customers; the two groups just don't talk. We certainly aren't recommending such a scenario but it does happen.

Whatever the cause, there is a temptation for developers to make their best guess and go on. The problem is that developers are naturally biased and they often assume customer behavior

that conforms to their own design. However, there are always other ways to think about the application, some of which may not mesh with the developer's view.

Therefore:

Create a SURROGATE CUSTOMER role in the project and fill that role with someone who will try to think like the customer. Treat the SURROGATE CUSTOMER like the real customer.

If the organization has human factors people, they make natural SURROGATE CUSTOMERS. Their emphasis may be on the human interface, but developing that interface is often much of the battle.

System test organizations are similar to SURROGATE CUSTOMERS, but there are important differences in intent. System testers tend to evaluate a product with respect to a specification in order to determine its readiness for market. Customers, real or surrogate, are interested in knowing if the product meets their needs and is easy to use.

Fellow developers tend to make poor SURROGATE CUSTOMERS. For the most part, developers think too much alike.

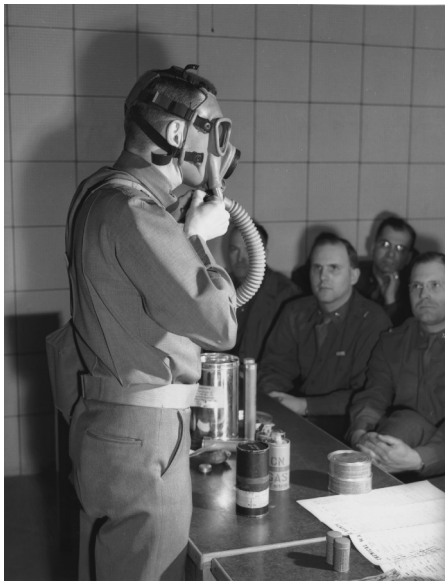
Of course, no SURROGATE CUSTOMER will ever replace a real customer. But surrogates can allow the project to move ahead in the absence of more concrete information. For more reading on the limitations of the SURROGATE CUSTOMER role, see [Constantine Lockwood 1999] and [Bramble 2002].

Perhaps the perfect ideal involves the developers themselves as customers or SURROGATE CUSTOMERS, if they can overcome the "nerdish groupthink" owing to their identity as developers. See CREATE RATHER THAN CONFORM (8.9) in the Quattro Pro for Windows case study.

Most organizations seat the SURROGATE CUSTOMER with the development team; in fact, this role is often a member of the development team. Consider instead seating developers at the customer site to avoid the problem described in the book *Contextual Design* ([Beyer Holtzblatt 1998], p. 34):

> Many IT departments avoid these problems by stationing IT developers with the customer organization. This certainly succeeds in making IT more responsive to the customer, but brings a loss of control. The developers easily become focused on short-term problems and solutions—they tend to become the local fix-it man. The structure of the customer's work and long-term possibilities for improvement are no more visible to IT developers than to the customer, and without this perspective they, like the customer, focus on the immediate and most visible issues. And they are stationed in a particular department, so cross-departmental issues are as invisible to them as to their customers. They are rewarded for producing quick fixes to pressing problems. The usual result is dozens of small applications, each solving a single problem, that do not work together to support the work coherently.

4.2.8 Scenarios Define Problem *



Discussing a worst-case scenario ...

How do you know a programmer is extroverted?
He stares at YOUR shoes when he talks to you.

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

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

There is a sense of natural business distancing and mistrust between customers and developers. Communication between developers and customers is crucial to the success of a system.

Therefore:

Capture system functional requirements as use cases.

It is obvious that use cases help increase understanding of requirements, but a less obvious aspect of use cases is that they help set boundaries of a problem. The latter concept became clear as one of the authors consulted with a group who was writing patterns of use cases. When I questioned a member of the group about what problem use cases solve, I got an unsatisfying answer. I probed deeper by asking how the situation would look if one didn't apply use cases,

and he responded, "[without use cases,] you wouldn't know where to start because the problem would be too broad." Interestingly, he had never thought about use cases as a tool to bound the problem until that point.

Use cases do not capture success scenarios alone: Instead, they capture all of the scenarios that the system must deal with. There is no such thing as an exceptional case; in other words, make the exception the rule. Interview enough constituencies to get full coverage of the expectations of users and other stakeholders. Use cases also can, should, and almost certainly must be augmented with nonfunctional requirements.

It is easy to see that capturing requirements as use cases is a good idea, but what does this have to do with organizations? One of the tensions in many organizations is that the developers are, well, geeks. Many don't have particularly good communication skills or, more precisely, aren't particularly interested in interpersonal communication. So it is difficult to communicate requirements to developers. However, scenarios work. So if you really want to ENGAGE CUSTOMERS (4.2.6), this pattern makes the job much easier.

The problem is now defined, and the architecture can proceed in earnest. You can use scenarios as a means of dialogue and requirements clarification with your users, particularly when building and demonstrating a system or subsystem prototype. For more on this concept, see CATALYTIC SCENARIOS in the DEMO PREP (A.5.9) pattern language from Todd Coram [Coram 1996].

Also read about the MERCENARY ANALYST (4.1.24), who captures scenarios and uses them for project documentation (both internal and external).

[Cockburn 2000] is one of the most acclaimed references on use cases. Also see [Goldberg Rubin 1995], which takes scenarios all the way to the front of the process preceding design, and [Hsia 1994].

4.2.9 Firewalls **

Nobody gets past this point without my permission!

"A manager should be like the sweeper in curling: The sweeper runs ahead of the stone and sweeps away debris from the path of the stone so that the progress of the stone will be smooth and undisturbed — does this sound like your manager?" [Gabriel 1996]

Unfortunately, heavy human use in this same area could lead to bear/human interactions which could injure humans and cause management actions against the bear. — Sign at an entrance to Boulder Mountain Parks, Boulder, Colorado

... an organization of developers has formed in a corporate or social context where they are scrutinized by peers and by funders, customers, and other "outsiders." Project implementors are often distracted by outsiders who feel a need to offer input and criticism.

✣ ✣ ✣

It's important to placate stakeholders who feel a need to "help" by giving them access to low levels of the project, without distracting developers and others who are moving towards project completion.

Isolationism doesn't work because information flow is important. But communication overhead increase nonlinearly with the number of external collaborators.

Many interruptions are noise.

Maturity and progress are more highly correlated with being in control than with being effectively controlled.

Therefore:

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."

The new organization isolates developers from extraneous external interrupts. To avoid isolationism, this pattern must be tempered with others, such as ENGAGE CUSTOMERS (4.2.6) and GATEKEEPER (4.2.10).

This pattern was present in both BORLAND QUATTRO PRO FOR WINDOWS (CHAPTER 8) and in A HYPERPRODUCTIVE TELECOMMUNICATIONS DEVELOPMENT TEAM (CHAPTER 9). See also the pattern ENGAGE CUSTOMERS (4.2.6), which complements this pattern.

GATEKEEPER (4.2.10) is a pattern that facilitates an effective flow of useful information; FIREWALLS restricts a detracting flow of (even potentially useful) information. You need a balance between the two. In the park in Boulder, people (customers) come to see nature, and bears are a part of nature. But if the customers interact too closely with the core contributors—to the point where they cause a distraction—things can get out of control. Developers need information, and they can take advantage of customer contacts and GATEKEEPERS to get the information they need. But they can also use managers as a shield. Furthermore, managers may need to step in to help developers who may be afraid to ask not to be bothered by customer contacts, or who are at risk of not fulfilling their own responsibilities if they are embroiled in customer matters.

Be warned that if the organization fills this role with someone motivated largely by personal power, the potential damage to the organization can be large. If other roles like GATEKEEPER maintain good contact with other organizations, communications are more likely to remain open, and FIREWALLS will more likely be called to account for self-serving actions.

As Sun Tzu notes, "He will win who has military capacity and is not interfered with by the sovereign" [SunTzu 1983].

4.2.10 Gatekeeper **

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

A project must develop good interfaces with the many outsiders with whom it interacts or with whom it should interact.

Most software development professionals—particularly programmers—are more comfortable interacting with their software and working with technology than working with people. Yet, isolationism doesn't work because information flow is important. On the other hand, communication has a cost: Communication overhead increases nonlinearly with the number of external collaborators. That overhead wouldn't be so bad if so many interruptions weren't mere noise. And an organization should be in control of its external interactions rather than letting the external interactions control it. Such control is a hallmark of organizational maturity.

Therefore:

One project member, a PUBLIC CHARACTER (4.2.17) with an engaging 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.

This pattern provides balance for the pattern FIREWALLS (4.2.9) and complements the pattern ENGAGE CUSTOMERS (4.2.6) (to the degree that customers are still viewed as outsiders).

GATEKEEPER and FIREWALLS (4.2.9) alone are insufficient to protect developers in an organization whose culture allows marketing to drive development schedules. However, this role can be made explicit in large projects whose budget and staffing profiles provide funding and support for such a role. But the role can also thrive informally in the margins.

GATEKEEPER is a pattern that *facilitates* the effective flow of useful information; whereas the FIREWALLS (4.2.9) role *restricts* the flow of detracting information. As described in FIREWALLS, self-serving people who work their way into this role can do much damage. It is probably healthier for the organization if this role is filled by someone who is not part of the management establishment. In such a case, it is more likely that peer support will sustain that person in the role, and it is more likely that the person will remain responsive to his or her constituencies. However, respected managers can also make great GATEKEEPERS.

The GATEKEEPER pattern has empirical value. In the discussion of this pattern at PLoP 94, many of the reviewers noted that creating a GATEKEEPER role had served their organizations well.

Engineers are lousy communicators on the whole; as such, it's important to leverage the abilities of an engineer who is an effective communicator when such a person 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), these subcultures should not be closed (MOSAIC OF SUBCULTURES, [Alexander 1977], ff. 42). Also see MAIN GATEWAYS ([Alexander 1977], ff. 276).

By analogy to Alexander's ENTRANCE TRANSITION, one might muse that the GATEKEEPER takes an outsider through any rites of passage needed to give that person intimate access to the development team ([Alexander 1977], ff. 548). In addition, GATEKEEPER can serve the role of "pedagogue," as in Alexander's pattern NETWORK OF LEARNING ([Alexander 1977], ff. 99).

Joe Maranzano notes that the same person often must fill both the MANAGER ROLE and GATEKEEPER role because of the relationships built with external people who need the information [Maranzano 1992].

If the GATEKEEPER (4.2.10) function starts taking on an aura of stability and legitimacy in its own right, it might point to the fact that key business issues cut across the existing organizations. Look at FUNCTION OWNER AND COMPONENT OWNER and UPSIDE DOWN MATRIX MANAGEMENT (5.1.19) as solutions that broaden the GATEKEEPER function to an organizational scope.

4.2.11 Self-Selecting Team **

Child coal miners, Circa 1910, the opposite of a Self-Selecting Team.

I had applied for a job in a different part of the company. It was forward-looking work on a small team. The manager was happy to take me, but it wasn't until the team had interviewed me that I got the job.

... SIZE THE ORGANIZATION (4.2.2) revealed the need for a small, select team. How do you staff such a team?

The worst team dynamics can be found in appointed teams.

There are no perfect criteria for screening team members. Yet broad shared interests (e.g., music and poetry) are an indicator that team players can work together successfully. Teams staffed with such individuals are often willing to take extraordinary measures to meet project goals.

However, when such interests are ignored or when team members are appointed, team dynamics can suffer, thereby greatly diminishing the productivity of a team.

Therefore:

Create enthusiastic teams by letting people select their own team members. Do limited screening on the basis of the team members' track records and broader interests.

Such teams often, though not always, come about of their own volition. Sometimes a PATRON ROLE (4.2.15) or other leader can seed the idea of such a team first as a rallying point for the formation of the team.

A SOLO VIRTUOSO (4.2.5) or APPRENTICESHIP (4.2.4) role may self-select a team. FORM FOLLOWS FUNCTION (5.1.11) can give such a team its structure, DIVERSE GROUPS (4.2.16) can help in the screening process, and temporary SELF SELECTING TEAMS can come together to work on PROGRAMMING EPISODES (4.1.19).

A SKUNKWORKS (4.2.14) is a special kind of SELF-SELECTING TEAM that comes together to share high risk on behalf of the organization.

Self-selection can, and should, happen at finer granularity than teams, too [e.g., see DEPLOY ALONG THE GRAIN (5.2.8)].

Such teams are different from "empowered teams." Research has shown that empowerment leads to communication locales that can become blindsided to the broader context of surrounding teams and that can unnecessarily narrow the communication channels between teams, though communication may increase within the teams themselves [Yates 1995].

One danger to be aware of is that an exclusive group of friends may build a team from their own numbers, failing to take advantage of other people's skills. The PATRON ROLE (4.2.15), however, can monitor these dynamics.

4.2.12 Unity of Purpose **

... the team is beginning to come together. Team members may come from different backgrounds and may bring with them many different experiences.

Many projects have rocky beginnings as people struggle to work together.

Often, people have different ideas about what the final product should be. In fact, the final product may well be a pretty fuzzy concept. Yet people must have a consistent view of the product if there is any hope of it getting done.

Each person is different and has unique views and opinions. They come with different backgrounds and experiences, and they must learn to work together.

It is important to get off to a good start—initial impressions, good or bad, tend to be lasting.

Therefore:

The leader of the project must instill a common vision and purpose in all of the members of the team. This "leader," whether a manager, the PATRON ROLE (4.2.15), or a customer advocate, should be someone who holds the team's respect and who has influence over the team's thinking. Gaining respect and influence requires overt action; you can't count on it

happening automatically. The leader should make sure everyone agrees on the answers to the following questions: What is the product supposed to do? Who are the customers, and how will the product help them? What is the schedule? Does everyone feel personally committed to the schedule? Who is the competition?

An important component of these team unification exercises is to identify the strengths of the team and to use these strengths as rallying points. The team thus identifies the challenges and competition and unites to overcome and surpass them, respectively.

As time goes on, the UNITY OF PURPOSE continues to emerge from ongoing dialogue within the team and with customers and other stakeholders. While the team leader primes the pump, team dynamics take over and keep things going.

The obvious result is that the team is working together rather than working at cross purposes. But a more subtle yet probably more powerful effect is what healthy team dynamics can do for the morale of the team. The best teams tend to feel that they are somehow better than others—and they work to prove it!

This pattern relates to some deep-seated principles and values of organizational health. There may be no more important single property of an organization than that its members have a shared vision that they are motivated to achieve. Communication—which receives the bulk of the attention in this book—is just a means to achieving that shared vision. UNITY OF PURPOSE, thus, is a deeper principle than even effective communication; communication is just a means to UNITY OF PURPOSE.

RELATED PATTERNS:

SHARED CLEAR VISION (A.5.25) ([Bramble 2002], p. 80) notes the importance of a clear vision in creating unity, from the point of view of writing use cases. SELF-SELECTING TEAM (4.2.11) outlines how a team should come together, though this guidance alone is insufficient to achieve UNITY OF PURPOSE. LOCK 'EM UP TOGETHER (5.2.5) helps achieve unity, particularly unity of architecture. A GATEKEEPER (4.2.10) also can help the team become more unified in establishing the requirements needed to ENGAGE CUSTOMERS (4.2.6). This pattern sets up COMPENSATE SUCCESS (4.2.25): it's much easier to compensate success when everyone knows what success means. And, while UNITY OF PURPOSE is important in galvanizing the team, effective team dynamics can develop only if every team member is also valued as an individual. HOLISTIC DIVERSITY (4.2.19) comes to play here.

4.2.13 Team Pride **

Problems worthy
of attack
prove their worth
by hitting back.

—Piet Hein (1905-1996)

... you are about to embark on yet another challenging project. The work will be technically difficult, or maybe you'll have a very short schedule. But at least you have some idea of what you want to do, which can begin to give you a sense of UNITY OF PURPOSE (4.2.12).

People are most successful when they feel good about their project and when they are confident. But there is a chicken and egg problem here: Confidence breeds success, but success creates confidence.

Pride perhaps goeth before a fall, but so does apathy.

Most software projects—sometimes even the fun ones—demand a lot of work. And the ones that aren't fun don't have much of a chance of seeing a victorious finish unless something pulls its people together and draws them on towards completion. The hard work feels even

harder because of short schedules. Such projects demand the best everyone can give, so motivation is often a key to success.

If people consider the work to be "just a job," the results will reflect this attitude.

Team success tends to become a self-fulfilling prophecy: everyone wants to work on a winning team, so teams can pick the best people. On the other hand, teams with low performance tend to be stuck with low morale. People don't join such teams willingly; instead, people are stuck with them.

So how do you improve the attitude of such team members? it is important to give a team a winning attitude right from the start.

Therefore:

Instill a sense of elitism into the team. Teams that have a certain arrogance tend to work hard and accomplish what is put before them.

Really, one cannot give someone else a sense of team pride. A sense of team pride must come from within. But there are many things you can do to help it come to pass:

- Start with a worthwhile problem. Team members are more likely to feel elite if they have a challenging problem to tackle. It is especially good if the problem involves new technology; nothing excites a bunch of geeks more than working with the newest stuff.
- Apply SELF-SELECTING TEAM (4.2.11). If the team self-selects, they will try to choose the best people.
- Find some important strength of the team and make that strength a rallying point. Teach the team that they have skills in a particular area. Be sure to find a *real* strength; people can easily see through a manufactured strength. The strength should be a technical strength; while a team might rally around the idea that "they party better than anyone else," this skill won't get the software written.
- Provide some explicit separation from other projects. This separation can be physical (separate the team from others), organizational, or informational (share secrets with the group). It can also involve exemption from some of the rules that everyone else must follow. Just make it clear that they are set apart from other groups.
- COMPENSATE SUCCESS (4.2.25).
- Hope that the parent company is doing well enough so that it isn't a concern. This author was once on a team that felt it was elite until the company started doing very poorly. The company woes diverted our attention and sapped our morale.
- Use FIREWALLS (4.2.9) to help instill the idea that "top teams shouldn't be bothered by bureaucracy."
- Unite against a common enemy, as with UNITY OF PURPOSE (4.2.12).

By itself, TEAM PRIDE does not guarantee the success of a project. But Boehm and others have pointed out that people are the key success element of any project, and TEAM PRIDE helps nurture and encourage them. Such pride may even be able to overcome poor overall morale in the company.

4.2.14 Skunkworks *

At the end of college, I was interviewed for a job with Lockheed Aircraft Corporations famed "Skunkworks" division. They had a huge skunk painted on the wall at the entrance to their work area. Everyone said the same thing to me: "We can't tell you what we do, but we sure have fun." I ultimately went to work elsewhere, but I occasionally wonder what it would have been like to work there. I don't know what work I would have been doing, but I'm sure I would have enjoyed it.

... organizations have the freedom to iterate and innovate early in the lifecycle of their major products. As a project matures, the context becomes rigid, and innovation becomes "forced" and may appear in the guise of "innovation programs." While these programs are good at encouraging the divergent thinking component of innovation, they rarely do a good job of encouraging convergent thinking. The result is that novelty, valued for its own sake, finds its way into mainstream development where it incurs costs and leads to results that range from indifferent to disastrous. "Home runs" are rare, and the net result is most often negative.

✣ ✣ ✣

A project must accommodate major innovations while also keeping an eye on risk. It is too risky to innovate too much in project development. Some projects have "innovation programs" that value divergent thinking, and the fruits of these efforts often make their way into development. It is only rarely that an organization honestly evaluates whether such ideas actually added value; the value, instead, is often taken on faith. For example, the latest technologies

are always held to have value in their own right; conversion to object-oriented (OO) design, to components, or to patterns is considered beneficial without even a second thought. Too often, these new ideas either have indifferent results or increase cost. They may decrease time to market or decrease cost, but if they decrease cost at the expense of time to market, then the overall effect is disastrous if time to market is the highest business priority. And, in fact, any new idea can increase both time to market and cost in ways that may never be noticed, in part because of the stock taken in the buzzword value of the idea.

Yet projects become dead if there is no way to incorporate paradigm shifts into them now and then.

Therefore:

Allow a limited-cost SKUNKWORKS to form [as a SELF-SELECTING TEAM (4.2.11)] in order to develop an idea outside the constraints of project development and to build confidence in the idea. Give the SKUNKWORKS organization ownership and credit for the idea.

The organization is sustained by strong FIREWALLS (4.2.9) that insulate it from the scrutiny of upper management and funders; in fact, the very existence of the SKUNKWORKS should be a secret. The idea is to keep the project off of management radar screens in order to foster the kind of innovation that leads to success before tradition and its constraints, as embodied in managers, can dampen innovation.

The success of the idea is assessed according to the fruits of the SKUNKWORKS effort: the ability of the resulting product to attract customers willing to invest time, money, or people in building the product or in otherwise furthering the idea. The product must *tangibly* show positive results that differentiate it from the mainstream product line. If it is to thrive over existing external *and* internal competitors, it must demonstrate distinguishing market superiority. Directly moving new ideas into existing business units rarely works. As a practical matter, this evaluation of success and the ensuing steps to act on it happen at unusual places in the management structure: at a higher level of management, in an organization that has venture funding, or in the marketing group, which can use the customer needs statements and customer commitments as leverage. However, the technology's chance of long-term success is much higher if the Skunkworks team includes developers who also have responsibilities in existing products. These developers can provide new development teams with ideas for new products or, they can be conduits for introduction of the new technology into development organizations. Therefore, this pattern also depends on a small set of interested developers having some limited amount of time to work with the SKUNKWORKS team in a GATEKEEPER (4.2.10) capacity.

The SKUNKWORKS organization itself rarely can take a product all the way into production. It usually lacks the infrastructure, and sometimes the skill set, to build a solid product. This phenomenon is at the root of the inability of many large companies to capitalize on their greatest inventions.

If the idea succeeds, the team should reap the benefits of developing the idea. The organization subsidizes some of the risk of the team under the sponsorship of a PATRON ROLE (4.2.15), so that the risk takers are guaranteed some minimum level of security even if they fail. However, these risk takers are not *guaranteed* the same level of rewards as people who succeed in lower-risk ventures [see COMPENSATE SUCCESS (4.2.25)].

This pattern is a bit different from BUILD PROTOTYPES (4.1.7). Prototyping is one strategy for running a SKUNKWORKS operation; however, a SKUNKWORKS project may just buy an existing product and integrate it with existing products or market it differently without doing any prototyping.

This pattern does not integrate with the other scheduling and organizational structures in the pattern language because it's a decoupled effort. The effort should evolve into a product over time and eventually incorporate patterns like SIZE THE SCHEDULE (4.1.2) and SIZE THE ORGANIZATION (4.2.2), but only after it's on its feet and has proven itself.

It is important that the SKUNKWORKS be organizationally separate from the mainline organization, which allows so-called disruptive technologies to flourish within the company. (For further information on disruptive technologies, see works by Clayton Christiansen, [Christianson 1997].)

Though it's clear how SKUNKWORKS fits into a large organization, it is useful to note that it can work on a smaller scale in small organizations as well. A couple of team members can develop innovative ideas "in the margins" as a side activity. Such opportunities may provide particularly good outlets for employees whose skills are strong enough that they seek challenges beyond those offered by day-to-day business.

4.2.15 Patron Role **

... the development organization has come to the point where DEVELOPER CONTROLS PROCESS (4.1.17), and now additional roles are being defined.

It is important to give a project continuity. But centralized control can be a drag, and anarchy can be even worse. However, most societies need a king/parent figure, and an organization needs a single, ultimate decision maker. The time to make a decision should be less than the time it takes to implement it.

Therefore:

Give the project access to a visible, high-level manager, who champions the cause of the project. The patron can be the final arbiter for project decisions, thus providing a driving force for the organization to make decisions quickly. The patron is accountable for removing project-level barriers that hinder progress and is responsible for the organization's "morale" (sense of well-being).

Having a patron gives the organization a sense of identity, and a sense of focus for later process and organizational changes. Other roles can be defined in terms of the patron's role. The manager role is not one of total centralized control, but rather one of championing the team. That is, the manager's influence usually does not include those developing the product itself, but instead includes those whose cooperation is necessary for the success of the product (e.g., support organizations, funders, and test organizations). This role also serves as a patron or sponsor and sometimes even as a corporate visionary.

We have observed this role being implemented by Philippe Kahn in QPW; Ravi Sethi and others in early C++ efforts at AT&T; a manager of a high-productivity Network Systems project at AT&T; and others at another multilocation AT&T project.

This pattern relates to the pattern FIREWALLS (4.2.9), which in turn relates to the pattern GATEKEEPER (4.2.10). Patrons are central to the success of SKUNKWORKS (4.2.14). They also can help arbitrate the membership of SELF-SELECTING TEAMS (4.2.11) to guard against exclusivity.

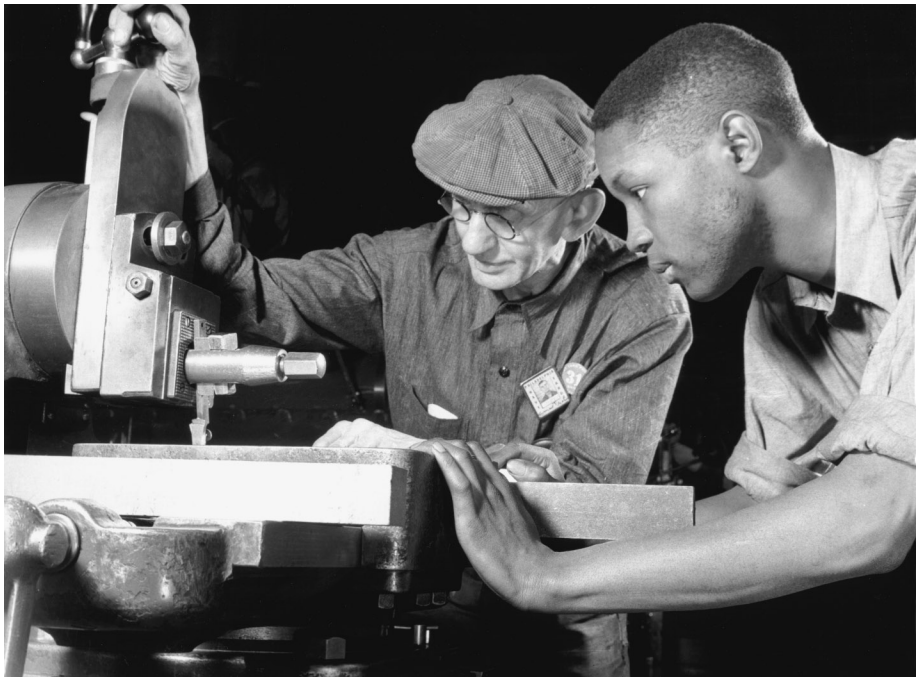
Block talks about the importance of influencing forces over which the project has no direct control [Block 1983].

In a JAD session ([Kendall2002], pp. 132–135), one of the key roles is that of a "tie breaker" who is usually a manager who appears only occasionally at the meetings.

The etymology of the term *Patron,* as described in a dictionary of medieval terms is instructive:

> 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 and pattern became differentiated in form and sense.
>
> ... 'The original proposed to imitation; the archetype; that which is to be copied; an exemplar' ... an example or model deserving imitation; an example or model of a particular excellence [Sane 1996]. aC. 1369 CHAUCER Dethe Blaunche 910 Truely she Was her chefe patron of beaute, And chefe ensample of al her werke.

4.2.16 Diverse Groups *



... a development team is forming, and birds of a feather tend to flock together.

Homogeneous teams that comprise too many of the same kinds of people easily fall into groupthink-like dysfunction.

Design is the act of changing the structure of the world. In software, it usually means changing the literature of an author who had encoded a solution in a programming language. That author usually remains as part of the community that retains an interest in the code [see CODE OWNERSHIP (5.2.13) and the combination of CONWAY'S LAW (5.1.7) and ORGANIZATION FOLLOWS MARKET (5.1.9), which implies that architecture follows market].

Change is a process that has several phases, starting with complacency, which is upset by an opportunity or realization of an oversight. The struggle to identify and to realize solutions culminates in deployment.

Different people are more comfortable with some parts of this process than with others. Some people are good at identifying problems, and others are good at the innovative processes of identifying solutions. Yet others are good at focusing on implementation. The variance in comfort comes from a variance in experiences and individual backgrounds and temperaments. These

variances are true even when programmers, in their role as designers who are making a change, are the same programmers who were the original authors of the code.

Therefore:

Consider temperaments and diverse experience backgrounds when assembling a team. This diversity sometimes corresponds with social classifications like age and gender, but more generally can be assessed on a personal level.

One source of variation is the variety of domains in the application itself [see DOMAIN EXPERTISE IN ROLES (4.2.22)]. There is an open question as to whether a SELF-SELECTING TEAM (4.2.11) prejudices a homogeneous group outcome; in any case, DIVERSE GROUPS can be a good audit of a SELF-SELECTING TEAM.

Another source of difference is the variation in roles. The vestigial pattern DIVERSITY OF MEMBERSHIP recommends building a requirements team from diverse roles:

> The team should include a developer, a user or user's representative, and a system tester (at least one of each). These individuals will work through the issues surrounding product requirements, often using small prototypes to identify the requirements and determine testing criteria. The user of prototypes can be closely tied to using use cases or similar usage scenarios as analysis and validation tools.
>
> One area in which this approach is especially useful is in the specification and design of the user interface. The developer creates mock-ups of the user interface, and the user and system tester examine them. In this way, this small subteam can go through many different designs of the user interface and select the best one [Harrison 1996].

See more about this kind of diversity in HOLISTIC DIVERSITY (4.2.19).

Sometimes teams form around mutual interests and talents in a domain, as in SUBSYSTEM BY SKILL (4.2.23), and diversity falls along other dimensions of interest.

Yet another kind of diversity is ethnic diversity. The oft-touted value of ethnic diversity is that it brings together people who can think about problems differently and from much different perspectives, which improves the chance of finding a good solution. But there are other subtle advantages. In a large multinational corporation, we found that each department had a few members of French national origin. The French people all ate lunch together, which provided a natural path of communication flow between departments.

One kind of person you want to include in the mix is a PUBLIC CHARACTER (4.2.17).

However, one must guard against stereotyping people (e.g., using personality instruments or other information to *limit* the roles of people in organizations). See [Kerth Coplien Weinberg 1998].

See the related pattern BALANCED TEAM (A.5.5) in [Bramble 2002].

4.2.17 Public Character *

... an organizational structure is emerging, both formally and informally, and frequent contact in the workplace cultivates friendships as well as a social context that begs for support of common social graces and functioning.

An organization is a social entity whose smooth functioning depends on more than professional relationships.

Much of what defines "culture" are the widely known but rarely spoken myths, tidbits, histories, and interpretations of stories about an organization and its people. However, most professional organizations are built around the exchange of more structured information in blatantly public forums: memoranda, meetings, explicit policies, and executive pronouncements.

Yet daily small pieces of information, details, and deep insights are the glue that hold the organization and its systems together. Furthermore, this information might include insights on shortcuts and other expediencies that serve the culture and its value system while falling short of the "letter of the law." The formal organization rarely has any organ that legitimizes the exchange of such information, yet such information is crucial not only to the smooth operation of the enterprise, but also to its very survival.

Such details include information that falls outside of the primary business goals, but which is nonetheless important to the support of the work environment (e.g., where to find a good place for lunch, how to find the boss when she's not in the office, and who knows how to fix the jam in the copy machine). It also includes metaknowledge about how to find the answers to certain kinds of information. For example, who would know how to find answers to questions about the web server machine? Who would know where to direct questions about personnel issues?

Therefore:

Ensure one or more people serve in the role of PUBLIC CHARACTER in order to help social processes both behind the scenes and at social events.

There may be sociotechnological role combinations. For example, an architect role might spend time passing information between development coordinators who otherwise wouldn't take the initiative to talk with each other [Coplien Devos 2000]. We wrote up this pattern as SHMOOZING ARCHITECT at Object Technology (OT) '99.

GATEKEEPER (4.2.10) and MATRON ROLE (4.2.18) are examples of PUBLIC CHARACTERS.

Jane Jacobs notes the following in *The Death and Life of Great American Cities*, [Jacobs 1961] p. 68:

> The social structure of sidewalk life hangs partly on what can be called self-appointed public characters. A public character is anyone who is in frequent contact with a wide circle of people and who is sufficiently interested to make himself a public character. ... His main qualification is that he *is* public, that he talks to lots of different people. In this way, news travels that is of sidewalk interest.

Jacobs goes on to say that, once the neighborhood recognizes a PUBLIC CHARACTER, people consciously share gossip with that person (e.g., regarding meeting dates and lost items) that they want propagated. A PUBLIC CHARACTER is a sort of living bulletin board, with highly advanced search capabilities.

One finds a similar function in the Maven role in *The Tipping Point* [Gladwell 2000].

In our experience, large software projects usually have at least one PUBLIC CHARACTER, and this person is critical to project success. When you want to know who understands the persistence layer, you don't ask the architects; they're too busy. Instead, you ask the PUBLIC CHARACTER, who won't know beans about the persistence layer, but who will know that Mary is a database expert and that she will either understand the persistence layer or will know who does.

One interesting form of PUBLIC CHARACTER is the Jester, or WISE FOOL (4.2.21). In medieval courts, the Jester was a person who could give advice and make fun of the king with impunity. The king was not obliged to follow the jester's insights; rather, these insights simply provided food for thought. Jesters can incite the organization to engage in introspection; again, part of their qualification is that they are public figures. Such a person might be instrumental in facilitating workshops using creative techniques, such as visual meetings, system envisioning concepts, and games. The jester might also report on user fears and expectations and act as a change agent. This role is also reminiscent of the "laughing uncle" configuration Bateson talks about in

his writings on Pacific cultures [Bateson 1958]. This uncle advised a child's father of feelings that the child might not have been able to convey to the father directly.

Project members are often penalized for being PUBLIC CHARACTERS. For example, some might say, "Oh, she never gets anything done, she's always gossiping." However, a PUBLIC CHARACTER is a vital part of an organization's ability to keep large projects connected and successful. In a number of cases, we have seen that the disappearance of a single public character caused a major turn in morale and culture in the organization, *to a much greater degree than the loss of a key technical person.* The role is essentially informal; a project manager can't successfully assign somebody to this role. Rather, the role is recognized and exploited when it is already present. Such recognition can help sustain the role.

If you see a team member who is always gossiping, consider whether the team member has become a PUBLIC CHARACTER. Ask the person a couple of team-related questions (e.g., "Where can I find out more about the garbage collection?" "Who understands the compiler tools"?). If the individual can handle these, as well as other more general questions (e.g., "Where's the best place to have lunch?" "How can I find Phil if he isn't at his desk?" "And what about... Naomi?"), then you've found your PUBLIC CHARACTER.

It is instructive to compare the PUBLIC CHARACTER role, GATEKEEPER role (4.2.10), and MATRON ROLE (4.2.18). In fact, the three roles are related, but different. The MATRON ROLE is concerned with nurturing the organization and is inward focused. On the other hand, the GATEKEEPER is outward focused, always looking forward to the next great direction for the organization to take. The PUBLIC CHARACTER is somewhat in the middle of these two roles, but separate from each. An ideal project has each of these roles filled by different people.

A good place for the PUBLIC CHARACTER to hang out is at THE WATERCOOLER (5.1.20).

4.2.18 Matron Role *

One of the members of my group was a woman named Anita. She was certainly technically competent, but I remember her more for the nontechnical things she did for the group. For birthdays, Anita was almost always the one who brought cakes, pies, or other treats to celebrate. Because she liked to cook, many treats were homemade; in fact, she occasionally brought in something just because she had tried out a new recipe. She did other things for the group too. She was often on picnic committees and helped to arrange "take your daughter to work" days.

Anita eventually moved on to another group, and our group has since been fragmented into other groups. But we still remember when we were a cohesive team, and Anita was a major part of the team.

... once a team is established, regular care and feeding is needed to maintain the unity of the team.

Teams do not survive simply because of the work they do. Some social activities are necessary to keep the team focused on the technical work.

"All work and no play makes Jack a dull boy," and the same holds true for teams. Unless teams play together some, they have trouble maintaining healthy interpersonal relationships, even in work situations.

But many people are not particularly adept at arranging social functions for their teams. This is particularly true in software organizations, which are dominated by introverts. In fact, some people are not even sufficiently aware of the importance such functions to be of any use in planning them.

Therefore:

Make sure that the team contains a Matron who will take charge of the social and interpersonal activities necessary to keep the team unified.

The Matron keeps track of birthdays and other occasions for celebration. The Matron is often willing to plan activities and usually serves as a member of party committees.

Note that you can't force this role on someone; some people are naturally Matrons, some are not. Therefore, you need to find one rather than manufacture one.

With a MATRON ROLE, the team is much more likely to be cohesive in good times and in bad.

Don Olson's PEACEMAKER (A.5.21) pattern ([Olson 1998], p. 168) is similar ([Rising 2000], p. 131):

> A peacemaker is a placeholder in an organization who tries to calm and hold things together until a leader can be found or a reorganization is complete. The peacemaker should be someone who is well liked but who is not necessarily technically proficient. Usually this individual has many years with the company, knows the political ropes, and can buy time for a team as well as the team's management.

MATRON ROLE is a broader version of the PEACEMAKER role.

The MATRON ROLE is usually a PUBLIC CHARACTER (4.2.17).

4.2.19 Holistic Diversity *

Even the manager pulls his weight in this small team, which is cooking up some new concoction.

... during the course of a project, groups of people begin to specialize. Teams are structured by specialty or by phase deliverables. This specialization leads to bureaucratic processes, lack of interteam communication, and a "throw it over the wall" style of development. As a result, teams don't trust each other, and product quality and efficiency suffer.

Development of a subsystem requires many skills, which can become a problem since people specialize.

Project development demands fast feedback, and fast, rich communication regarding decisions. Feedback slows with distance (e.g., room, floor, building, or city) and medium of expression (e.g., interactive face-to-face discussion, video teleconferencing, or writing).

Multiple skills are needed to develop a piece of the system, particularly the user functions, and it is hard to find people with those multiple specialties. In addition, people tend to specialize and even protect their own unique skills from others, as a natural self-preservation mechanism. Many teams thus tend to specialize.

So a project requires multiple skills, which tend to reside in separate teams. But this arrangement is not optimal. People within a team are more likely to help each other, whereas people on different teams are more likely to blame each other. Communication across teams tends to be inefficient and incomplete.

The obvious approach is to create one giant team for the project. This should solve the problem, right? But if the team is too large to fit comfortably in one room, it tends to fragment naturally—along lines of specialization.

Therefore:

For each function or set of functions to be delivered, create a small team (two to five people) that is responsible for delivering that function. That team can be seeded with or can evolve specialists in requirements gathering, user interface design, technical design and programming, databases, and testing. Evaluate the team as a single unit so there is no benefit to hiding within a specialty. Arrange the team location so the team members can communicate directly with each other, instead of by writing. The team has no internal documentation requirements, although they do have documentation responsibilities to the rest of the project. However they choose to split up their work is their choice.

Note that this leads to organizing teams along architectural lines in accordance with CONWAY'S LAW (5.1.7). As a result, it is necessary to coordinate the teams to get consistency of deliverables (e.g., requirements document, user interface design, and software architecture) across teams.

Beware of making teams too small. If the team size is one person, that person will have difficulty in mastering all of the specialties and in changing mental context to perform well in the different specialties (meetings require a different temperament and more concentration than designing OO frameworks). [See SOLO VIRTUOSO (4.2.5).] On the other hand, if the team size is too large, communication latency increases.

See also OWNER PER DELIVERABLE (A.5.19), which ensures that somebody owns each function, class, and required deliverable.

This pattern is similar to DIVERSITY OF MEMBERSHIP [Harrison 1996], which is used to ensure that requirements gathering teams include users.

Jim McCarthy [McCarthy 1995] wrote FEATURE TEAMS with much of the same intent.

It is hard to find single individuals who can master needed specialties and change work contexts as needed. Creating a small, co-located, mixed-specialty team with no written deliverables between them increases the communication bandwidth between people, while letting the individuals develop their strengths. Rewarding them as a team keeps them motivated to help each other deliver, rather than to hide behind their specialty.

There is a tight connection between the specialties. A designer or programmer may discover something that reveals that the requirements are more difficult than previously thought. The analyst may have a flawed view of the business, though the final code must be a valid business model. The suggested user interface may be impractical to implement, or perhaps the user interface designer knows best how to implement it. Putting diverse people on the same team speeds up the feedback loop between programming and requirements. Separating those same people and putting written deliverables between them slows that feedback.

Alistair Cockburn tells of his experiences with a project:

> Project Winifred was initially structured by function, which produced the trouble that many people were altering one class at any moment in time...
>
> It was next structured by phase deliverables and requirements. Analysts were separated from designers and programmers. The analysts produced ineffective models, communications between the people became sluggish, the analysts and programmers looked down on each other, and the analysts' designs did not match the final system design (the programmers ended up designing it as they needed to make it work).
>
> There was a very brief period of "everyone does everything." It did not last long because the mental load was too great on each person trying to do everything, and people rapidly fell into the specialties they could handle.
>
> The fourth, and successful arrangement, was HOLISTIC DIVERSITY [4.2.19]. Those who could do the requirements gathering and analysis went to meetings, interviewed people, and investigated interfaces and options. They communicated the results rapidly, face-to-face, with the people who navigated the class library and designed classes and frameworks. A function team consisted of a combined requirements gatherer/analyst with two to four programmer designers.
>
> The team used JUST DO IT to move rapidly through the design. They had no internal deliverables, but created the deliverables as required by the project for interteam communication and maintenance. Most of the communication within the team was verbal. They talked several times a day, either in one-hour mutual-education sessions, or in small, several minute interchanges to mention a recent discovery. This amount of communication could not have been handled through formal deliverables.

See also DIVERSE GROUPS (4.2.16).
This pattern was originally written by Alistair Cockburn [Cockburn 1996].

4.2.20 Legend Role *

Baseball legends George Sisler, Babe Ruth, and Ty Cobb

The hero Westley had returned in the guise of the Dread Pirate Roberts. He explained to Princess Buttercup that he had been trained by the previous Dread Pirate Roberts: "One day Roberts pulled me aside. I'm not the Dread Pirate Roberts, said he. And the man before me wasn't either. Then he explained that the name was important. You see, no one would surrender to the Dread Pirate Westley" (from The Princess Bride [Princess Bride 1987]).

... over time, certain people really excel at their jobs. They become masters, and they take on many important jobs on the project.

Certain individuals take on so many jobs and become so important to the project that when they leave, the project is in more than just serious trouble.

These individuals are generally the elder statesmen and women on the project. They have been around longer than most anybody else, and their depth of experience is invaluable. But because of their age, they are the ones most likely to retire.

Not all people are masters. These people are the ones who tend to pick up extra work and the associated expertise, so their absence is felt all the more. In fact, it seems like it would take two or more people to fill each of their shoes.

Therefore:

Name a role after the person, and make it an honor to fill that role. People will want to emulate the legendary person and to do just as good a job.

In many cases, the role named after the person will naturally emerge. Then it is a matter of formalizing the role a bit and filling it when the legend retires.

There *must* be training provided for the person filling the role. Ideally, it is offered by the LEGEND ROLE as part of the process of turning over the role to the new person. This training is as important as the naming of the role itself.

A software company we analyzed had a role named "Simon." They told us that Simon had been a key player on the project and had done seemingly everything. They used his name for the role involving the jobs he had done.

Some corporate cultures are built around archetypes. For example, electric power companies are built around the heroic acts of linemen working during threatening weather.

Emulation can be encouraged with an award. This author wrote some patterns of shepherding. Later, the Neil Harrison Shepherding Award was established, which encourages shepherds to excel at the work they do.

Filling the LEGEND ROLE helps to maintain project knowledge and expertise over time, as well as helping to keep a MODERATE TRUCK NUMBER (4.2.24). Note that legend roles will fade over time, which is generally all right.

There is a subtle but important difference between having a legend *role* and having the actual legendary *person* on staff. In writing about his CULT OF PERSONALITY pattern, Don Olson ([Olson 1998], pp. 154–155) offers this advice:

> A tight schedule, poorly defined requirements, uneven distribution of skills among the development team, and new technologies have put a project in jeopardy. To save the day, bring in a legendary figure among the developers to take over the lead. Team members who are not impressed may need removal or reeducation.

LEGEND ROLE looks longer term and intends to be an inspirational rather than a remedial pattern. CULT OF PERSONALITY can work if the legendary figure offers true leadership and develops growth in the team, but then it is no longer a "personality cult" in the vernacular sense. It is dangerous for a team to develop too much dependency on a single power figure, because the team has difficulty adjusting to a new communication structure, authority and control structure, and culture when the legendary figure is gone. Also, the LEGEND ROLE could become a bottleneck under these situations [see DISTRIBUTE WORK EVENLY (5.1.13)].

This overreliance, in fact, was noted by Alistair Cockburn as being a problem in the XP-based [Beck 1999] C3 project, where he characterized XP as a high-discipline methodology and

likened it to Humphrey's Personal Software Process [Humphrey 1995]. The following commentary comes from the WIKI WIKI Web [Wiki-Discipline 2001].

> I consider XP a HIGH DISCIPLINE METHODOLOGY, one in which the people will actually fall away from the practices if they don't have some particular mechanism in place to keep them practicing. Ron [Jeffries] is that mechanism at the moment. Should (when) Ron leave, then unless he is replaced in his role, I quite expect to see the team not following the practices properly in less than 6 months.

Ron did leave the project, and the following discussion appears on the CThreeProjectTerminated Wiki page [Wiki-Terminated 2001]:

> *... It wasn't "to live" it was to stop following all of the practices.*
>
> - "unless [the coach] is replaced in his role, I quite expect to see the team not following the practices properly in less than 6 months. I think that is a fair test of a HIGH DISCIPLINE METHODOLOGY. — Alistair Cockburn"
> - "I'm no longer on C3 full time. Alistair's six-month clock has started. — Ron Jeffries 6/25/99"
> - "As of the first of February, 2000, the C3 project has been terminated without a successful launch of the next phase."

The coach, in fact, does figure strongly in the XP organization. The coach is "responsible for the process as a whole" and sometimes must intervene to the point of "rudeness ([Beck 1999], pp. 145–146)." However, XP as published recognizes both the danger and difficulty of interventions that are overly direct and immediate. But our study of several projects claiming to use XP practices found strong elements of a personality cult. In one case, the team leader on XP project at an insurance company became more assertively involved when the project got behind schedule (the project dutifully and effectively used the XP planning game).

If, instead, the legendary figure consults with the team, with the aim of helping the team members to grow, this approach can be effective. See [Weinberg 1986] for ideas.

4.2.21 Wise Fool *

I marvel what kin thou and thy daughters are: they'll have me whipped for speaking true, thou'lt have me whipped for lying; and sometimes I am whipped for holding my peace.

—The fool, King Lear, act 1, scene 3

... a team has been established and is functioning. It is faced with a continual barrage of technical and nontechnical challenges, about which it must make decisions.

Interpersonal dynamics often discourage good ideas from being aired and bad ideas from being weeded out.

Two dynamics are at work here, depending on the persons involved. Authority figures are often unchallenged: You might be reluctant to challenge your boss because of the perceived danger to your continued employment. People are also loathe to challenge the word of a respected elder in the organization for slightly different reasons. But this reticence tends to allow bad ideas promoted by authority figures to promulgate without sufficient challenge and discussion.

The other dynamic is the group itself. It is difficult to stand up to an entire group to challenge an idea. These days, such troublemakers are rarely tarred and feathered, but they might be ostracized or labeled as poor team players.

Yet somebody needs to be the catalyst of occasional group introspection. Someone needs to shout a warning when the group heads in the wrong direction.

Therefore:

Nurture the role of the wise fool who can raise uncomfortable truths with impunity.

The WISE FOOL asks the questions that may be unpopular or that seem politically risky, but these questions make the team pause and reexamine decisions. Often, many people want to ask the same question, but they do not dare. The WISE FOOL displays a mix of insight, candor, and foolhardiness.

The WISE FOOL is legendary. The most famous WISE FOOL may well be found in the story of the Emperor's New Clothes. In the story, only a small boy has the courage to point out the obvious.

The WISE FOOL is much like a PUBLIC CHARACTER (4.2.17). However, the PUBLIC CHARACTER makes the group function smoothly, whereas the WISE FOOL focuses mainly on the (mainly technical) outputs of the group. Like the PUBLIC CHARACTER, the WISE FOOL is not designated, but emerges. A WISE FOOL, though known for lacking tact, is usually highly respected technically and may be (or become) a LEGEND ROLE (4.2.20). They usually eschew managerial opportunities and may even show disdain for management. An acquaintance of the author was once honored with the following words: "In the face of management opposition, he charged ahead and did what was right."

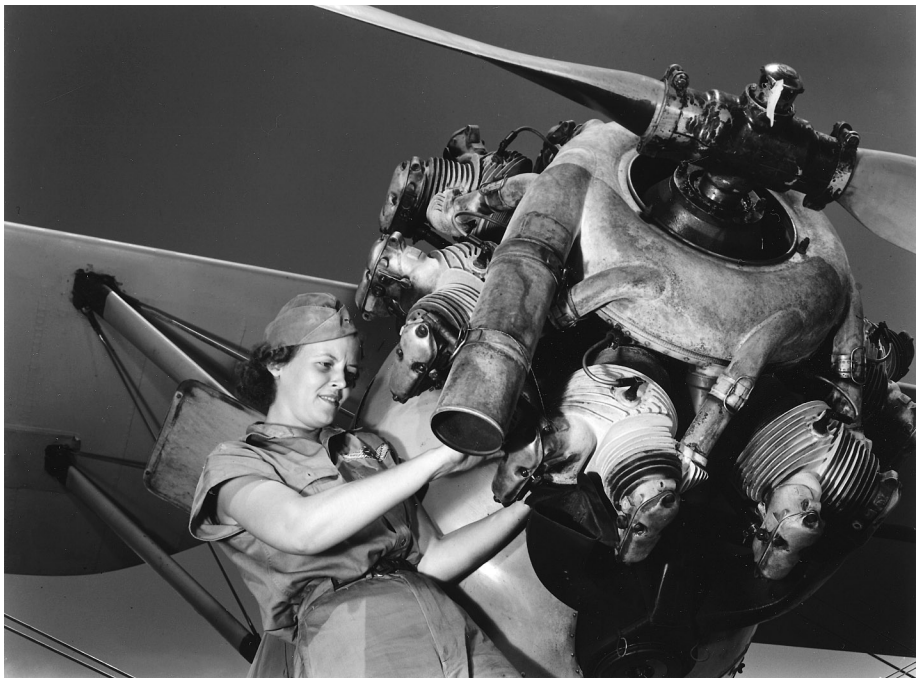
Some organizations recognize the need for a WISE FOOL. One organization we studied even included a role called "Agitator."

A WISE FOOL needs to recognize the difference between asking legitimate questions and complaining incessantly. Questioning things that one has no control over is often construed as whining. Too many such questions can lead the court of public opinion to demote a WISE FOOL to a Whiner rather quickly.

Organizations that have the good fortune to have a WISE FOOL in their midst are likely to make fewer wrong decisions than other organizations. However, WISE FOOLS may not receive the recognition they deserve and may be perceived as troublemakers, a scenario that is slightly reminiscent of SACRIFICE ONE PERSON (4.1.22). Managers should be sensitive to this possibility and make sure that WISE FOOLS are supported.

Note that the key here is that the organization itself must be willing to accept criticism from within. There will always be people around willing to fill this role, but only healthy organizations benefit from their insights. In fact, it often doesn't come naturally even to healthy organizations. Some organizations within Siemens, for example, hold workshops to help create a culture where people can speak out [Ackermann 2002]. However, unhealthy organizations may ignore or, even worse, actively suppress criticism. This climate of fear of speaking leads to widespread cynicism. In such cases, a few WISE FOOLS will refuse to be silenced and will become whistleblowers. When they report illegal conduct to authorities, they may even need laws to protect their actions.

4.2.22 Domain Expertise in Roles **



Naval air base, Corpus Christi, Texas. A top notch mechanic, Mary Josephine Farley, expertly rebuilds airplane engines. Although she's only twenty years old she has a private pilot's license and has made several cross country flights.

... you know the key atomic process roles [FORM FOLLOWS FUNCTION (5.1.11)], including a characterization of the developer role.

Matching staff with roles is one of the hardest challenges of a growing and dynamic organization. All roles must be staffed with qualified individuals. Just as in a play, several actors may be assigned to a single role, and any given actor may play several roles.

You'd like to use domain-inspecific qualification criteria like college grades or years of experience to qualify people for jobs. Such an approach gives the project flexibility in staff allocation and it helps it avoid being overly dependent on individual skill sets and experience. In short, the hope that such criteria might work provides project managers with a basis for keeping the project from becoming overly dependent on certain individuals who may leave or who may hold the organization hostage to gain higher salaries or to see their own policies implemented unilaterally. Nonetheless, successful projects tend to be staffed not with people

who possess textbook qualifications, but instead with people who have already worked on successful projects.

Spreading expertise across roles complicates communication patterns. It makes it difficult for a developer or other project member to know who to turn to for answers to domain-specific requirements and design questions.

Therefore:

Hire domain experts with proven track records, and staff the project around the expertise embodied in their roles. Teams and groups will tend to form around areas of common domain interest and focus. 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.

Organizations can benefit from having local gurus in all areas from application expertise to expertise in methods and language.

This pattern is a tool that helps ensure that roles can be successfully carried out. It also helps make roles autonomous. Empirically, highly productive projects (e.g., QPW) hire deeply specialized experts. OLD PEOPLE EVERYWHERE ([Alexander 1977], ff. 215), talks about the need for the young to interact with the old. The same deep rationale and many of the same forces of Alexander's pattern also apply here.

This pattern also provides a systems principle that one finds in software development [Lea 1995].

A seasoned manager writes, "The most poorly staffed roles are System Engineering and System Test. We hire rookies and make them System Engineers. (In Japan, only the most experienced person interacts with customers.) We staff System Test with 'leftovers' after we have staffed the important jobs of architecture, design, and developer. [Anon 1997]"

Some roles [DEVELOPER CONTROLS PROCESS (4.1.17), MERCENARY ANALYST (4.1.24), ARCHITECT CONTROLS PRODUCT (5.2.3), and others] are prescribed by their respective patterns.

If expertise becomes too narrow, the organization is at risk of losing key expertise if a single person leaves, is promoted, etc. Temper this pattern with MODERATE TRUCK NUMBER (4.2.24).

Domain experts can naturally come together in PROGRAMMING EPISODES (4.1.19). The pattern APPRENTICESHIP (4.2.4) helps maintain this pattern in the long term. DIVERSE GROUPS (4.2.16) is, in some sense, a more general version of this pattern.

See also SUBSYSTEM BY SKILL (4.2.23) and UPSIDE DOWN MATRIX MANAGEMENT (5.1.19).

4.2.23 Subsystem by Skill *

... an organization of developers exists. They have different skills and specialties, but there is not yet any structure in the organization, or in the system architecture that reflects such specializations or interests.

Birds of a feather flock together. By CONWAY'S LAW (5.1.7), you want the architecture and the organization to match each other. Yet there are many possible principles of organizing both the software and the organization that builds it. One structure relates to domain knowledge and the system architecture. There is also a business structure and a geographic structure, as found in ORGANIZATION FOLLOWS LOCATION (5.1.8). But in ORGANIZATION FOLLOWS LOCATION, each location is largely autonomous and has its own organizational decisions to make, so the issue remains as to how to modularize the organizational structure locally. ORGANIZATION FOLLOWS LOCATION conveys global constraints that relate to business priorities and concerns; CONWAY'S LAW offers guidance in the large, but doesn't extend as well to the fine structure at the group level. And that structure—the primary low-level structure of the organization—relates to the subsystem structure. We need to enhance CONWAY'S LAW with a set of partitioning criteria.

Therefore:

Separate subsystems by staff skills and skill requirements.

This pattern, a refinement CONWAY'S LAW, indicates the criterion by which the structures of the organization should be aligned with those of the product.

People skills tend to be relatively stable over time, so this pattern protects organizations against shifts in staff.

The variation protected against here is the variation in staff skills over time. On a small enough project, a few people may have multiple skills that enable them to mix user interface (UI) design with infrastructure design and domain design. Unhappily, their successors may not possess these diverse skills, which makes system evolution more difficult and costly.

On larger projects, many people are more likely to have single skills and specialties. If their code is intermingled, two expensive difficulties accrue: getting different people to learn to understand each other and to come to common decisions and resolving the same system evolution difficulty as faced on smaller projects.

Separating specialties into different subsystems lets the team focus on their special issues in their own special vocabulary, lets their successors see those issues in isolation, and makes the project easier to staff, since the staff do not need to be so multidisciplinary. Once the subsystems are identified, various forms of teaming may be used to develop them.

The pattern, of course, should be applied in moderation, for too many subsystems leads to complex, slow software. And too fine of an organizational structure is unwieldy and cumbersome.

Note this pattern's relationship to DOMAIN EXPERTISE IN ROLES (4.2.22). This pattern removes one degree of freedom in DIVERSE GROUPS (4.2.16).

Related subsystems may be connected, while still providing a degree of independence between teams by using STANDARDS LINKING LOCATIONS (5.2.12).

UPSIDE DOWN MATRIX MANAGEMENT (5.1.19) is one way of handling SUBSYSTEM BY SKILL.

Alistair Cockburn offers the following analysis of the relationship between HOLISTIC DIVERSITY (4.2.19) and SUBSYSTEM BY SKILL (4.2.23):

> HOLISTIC DIVERSITY is aimed at streamlining communication: For each function or set of functions to be delivered, create a small team ... evolve specialists in requirements gathering, UI design, technical design, ... Evaluate the team as a single unit. Arrange the team size and location so they can communicate directly. You will have to coordinate the teams to get the ... UI design, software architecture and so on consistent across teams.
>
> SUBSYSTEM BY SKILL is aimed at protecting the system against "variation in staff skills over time"— I thought of it primarily as a software design pattern, rather than a project management pattern, which is why I hadn't thought of them together. "Many people are more likely to have single skills and specialties ... Separate their specialties into different subsystems."

What happens if you put the two together? You get the team structuring I described in my book [Cockburn 1998, p. 88] for a 40-person project: function teams using HOLISTIC DIVERSITY, infrastructure teams, ARCHITECTURE TEAM, and technology teams. The UI gets its own subsystem, the domain model gets it own subsystem, the database gets its own subsystem ... and now you have to use HOLISTIC DIVERSITY to get all the parts put back together to make a working system. [Put] expertise from each specialty on each team (some team members bring more than one specialty with them). And you also have to run a UI group across function teams to get consistent UIs; a persistence and a domain group similarly to get consistency there, too. The software ends up partitioned by skill. So you work extra hard to see that the teams don't get similarly segregated. This is the stuff that my book covers, in its tiny way.

What happens if you don't put the two together? If you don't do SUBSYSTEM BY SKILL, then you get UI, domain, persistence, networking code all mixed together. Yuck, but that's well known. Why have we long separated these things? Because they change independently or because they capitalize on different specialties? or we have those specialties because they change independently or they change independently because we have those specialties?

I don't know and won't guess.

What if you don't do HOLISTIC DIVERSITY? Then you get a room full of UI designers, another room full of domain modelers, another room full of persistence designers / DB [database] designers, etc. I think we have all seen enough of this and its negative consequences. I am, by the way, currently in an organization separated this way, and [am] trying to get the people, who sit only steps apart, to talk to each other on microteams.

So I think we need both: a project management pattern and a software architecture pattern that work together [Cockburn 2000].

4.2.24 Moderate Truck Number

In an insurance company we studied, the project scheduled some of its release dates around the vacation times of a small number of key staff. While such planning is much better than constraining vacation times to the release schedules, it would have been better if the project had been less dependent on those employees. As should have been predicted, the release dates slipped and interfered with the vacation dates anyhow.

... you have built an organization around specialists whose background and training match the expertise required by the application and market, the DOMAIN EXPERTISE IN ROLES (4.2.22).

A project cannot become overly dependent on any small number of individuals. It's important to have specialization. No amount of general accomplishment can compensate for experience. And this experience is not embodied in any abstract concept of roles and is seldom found in any supporting document or knowledge base that a plug-compatible-interchangeable developer could leverage. Instead, the expertise is most often embodied in a living human being who can make choices.

Such human beings may make unpleasant choices, such as leaving the organization for another company. Or they may make silly choices, like walking out in front of a truck at a busy intersection, never to return to the project again.

And life may make choices for such individuals, such as giving them prospects for promotion. Sadly enough, there is a high correlation between individual's perceived expertise and the chances that a company will offer them promotion to optimize the chances for the Peter Principle to have its way. Or, another project within the organization may take them away.

It is a risk if your project depends too heavily on such individuals for their singular knowledge. You know you're in trouble if your project schedules release dates around their vacation times.

Yet, it's still important for individuals to possess areas of expertise in order to reduce the need for communication between individuals regarding decisions within a certain business area. That helps ensure that the right experience is brought to bear in decision making.

And, it's important to recognize that *everyone* brings some expertise to the table. If everyone were the same, there would be useless redundancy in the organization [see DIVERSE GROUPS (4.2.16) and HOLISTIC DIVERSITY (4.2.19)].

However, not everyone can know everything. Being a true expert in an area requires all of one's attention, and it is difficult to sustain multiple areas of expertise.

Define the *truck number* as the number of people in the organization who have unique *critical* domain expertise. You don't want the truck number to be large, because that means that the probability is large that the loss of any given team member would mean the loss of critical expertise. The risk would be too high. Yet, it's also impossible to make the truck number very small (and it's almost impossible to make it zero). Even if you could make it small, you probably wouldn't; if it were one, then everyone but the critical resource is intellectually redundant. In other words, by some rationale, all of the other members of the organization could turn into overpaid worker bees or software assembly-line workers.

Therefore:

Keep the truck number low, thus retaining a small number of key experts with unique knowledge. Build a culture of shared knowledge that increases the breadth of knowledge over time, particularly for knowledge that easily can be codified, taught, or otherwise conveyed.

How do you build such a culture? One way is to use DEVELOPING IN PAIRS (4.2.28). Another way is to make sure the experts rub shoulders with the mere mortals [use ARCHITECT ALSO IMPLEMENTS (5.2.10)]. Of course, you retain a nonzero truck number by keeping the architects from becoming mere mortals themselves [see ARCHITECT CONTROLS PRODUCT (5.2.3)].

Cross-training can be an effective technique for sharing knowledge. In particular, APPRENTICESHIP (4.2.4) is an effective form of cross-training. However, some of the deepest forms of knowledge and gut feeling cannot be conveyed from an expert to an apprentice.

A pattern language of the organization's key competencies can provide some relief for experts and can reduce the risk to the organization. Collect patterns from domain experts.

The goal is *not* to level the playing field. You still need DOMAIN EXPERTISE IN ROLES (4.2.22). It is too expensive (in time and talent) to guard against any possible staff loss by completely replicating talent. You want to implement enough cross-training to control the costs of recovery from losing a person. Trying to spread expertise too broadly will, in fact, just dilute the overall expertise by detracting from each expert's focus.

The Truck Number is a measure of the vulnerability of an organization. It's usually pretty easy to calculate it: Just ask yourself, "Which people in my project can we absolutely not do without?" It's likely that several names immediately come to mind. These people are the key architects, programmers, or perhaps even testers. And they are critical in part because they know things that others don't. So we try to get them to share that knowledge with the rest of the team.

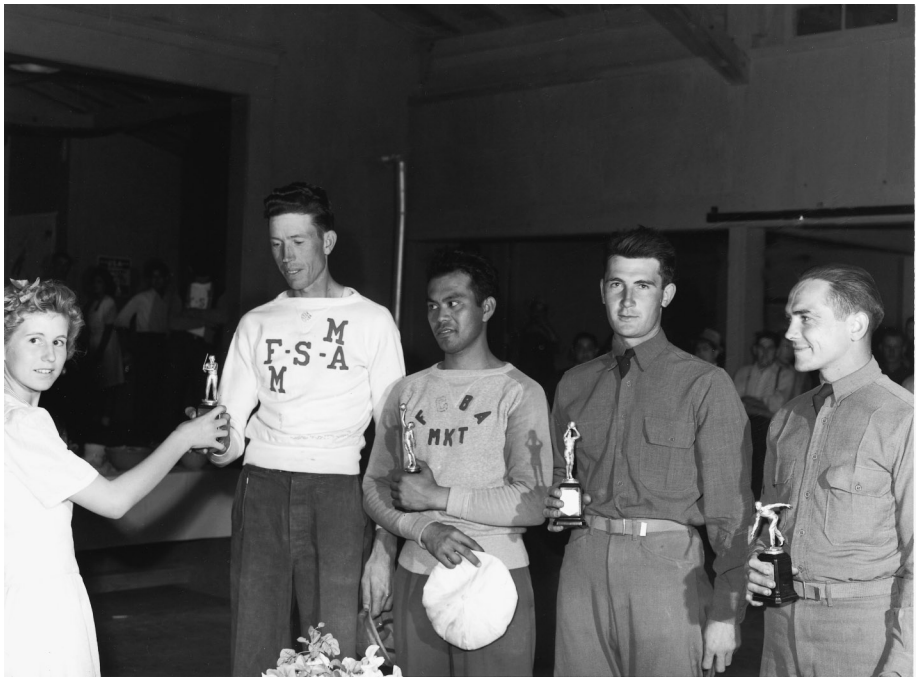
Note that although we speak of the Truck Number as a number, it has a subjective qualitative aspect to it as well. In other words, not all experts are created equal. The loss of some experts may cause serious problems, but the loss of others may be absolutely devastating!

One of the authors once studied a small software company. While the company and its (single) product looked good, one particular employee seemed to be unusually dominant. If he were to leave, the company would be in serious jeopardy. Unfortunately, he left, and the company suffered greatly. The moral: Watch such individuals closely, and make sure they continually share their expertise.

Why doesn't DEVELOPING IN PAIRS (4.2.28) solve the problem completely? It certainly helps, but people are still individuals who possess different skills. A pair in which each member is good at something different is greater than the sum of the individuals.

As with any risk reduction activity, reducing the Truck Number is an exercise in trade-offs. You may find that duplicating the expertise of certain people just isn't cost—or time—effective. So you live with the risk. Maybe you try to reduce it in other ways, such as creating incentives for those people to stay on the team [e.g., see COMPENSATE SUCCESS (4.2.25)].

4.2.25 Compensate Success **



When I was in fourth grade (about 9 years old), we had a spelling test every Friday. Our teacher told us that if everyone got a perfect score on a spelling test, she would bring each of us a candy bar the following Monday. We were excited about the prospect, but as time went on it seemed that our class might never earn our candy bars. Some people just couldn't seem to spell. Jimmy was probably the worst speller of all. He typically missed about half of the words. There was no hope for us with him in the class.

But one week the words were particularly easy. In the practice test on Wednesday, everyone except Jimmy got all of the words right. And Jimmy missed only four words. The anticipation in the class was electric, and we all gave special help and encouragement to Jimmy. On Friday, when everyone got a perfect score, it was hard to tell whether we were more excited about the candy bar or about Jimmy's success.

... a group of developers is striving to meet tight schedules in a high-payoff market. It is important to reward individuals in a way that motivates them to do things that achieve business objectives that are in line with the value system of the enterprise.

Successful projects remain successful by rewarding behaviors that lead to success.

Schedule motivations tend to be self-fulfilling, and a wide range of schedules may be perceived as equally applicable for a given task. Furthermore, pre-ordained schedules are poor motivators.

Some organizations count on altruism, but selfless, egoless teams are quaint, Victorian notions.

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

You need to reward both teams and outstanding individuals. Yet, disparate rewards that motivate those who receive them may frustrate their peers.

You need both to reward solid workers and risk takers. However, from an economic perspective, you need to manage the risk of any investment in speculative work. And if speculative work fails and the contributors are rewarded according to performance, the organization will have a disincentive from embarking on future risk-taking projects.

Some contributions are difficult to quantify, such as those of the catalyst (see [De Marco Lister 1976]) who facilitates communication between team members and perhaps helps morale.

Therefore:

Establish lavish rewards for individuals who contribute to successful make-or-break projects. The entire team (social unit) should receive comparable rewards to avoid demotivating individuals who might assess their value by their salary relative to that of their peers. Extremely valuable team members might receive exceptional awards that are tied less strongly to team performance.

A celebration is a particularly effective reward [Zuckerman Hatala 1992].

As a result, you get an organization that focuses less on schedule [See SIZE THE SCHEDULE (4.1.2)] and more on customer satisfaction and systemic success.

In most enterprises, you do not want to reward risk-taking in a way that encourages people to take risks that don't serve the long-term viability of the enterprise. The reward should always be more focused on meeting the organization's goals than on the manner in which the goals are met. If the organization's job is to produce a product, then reward people for what they do in support of delivering the product. Sometimes this support involves an element of risk-taking, and to that degree risk-taking should be rewarded. However, you want to remove *obstacles* to risk-taking in order to allow people to take appropriate risks motivated by a desire to meet organizational objectives, rather than for the sake of having taken a risk. See more about this concept in SKUNKWORKS (4.2.14).

Similarly, most software development organizations shouldn't encourage people to seek crisis situations as opportunities to make the contributions that will receive the highest reward. Doing so almost guarantees that the project will become crisis driven. Some jobs are legitimately built around a hero culture, such as (real-world) fire fighters and their figurative namesakes inside software projects, but these jobs are the exception rather than the rule. Be sure to reward what the organization values, knowing that people will tend to do what they are rewarded to do.

Similarly, it can be problematic to reward those who work for the sake of the work ethic alone. Reward accomplishments more than hard work; there should be no prize for the most hours worked. Paul Bramble relates the following [Bramble 2003]:

> Working for stock options that could be expected to turn into $12 million was a horrible experience. And having peers with similar expectations only made it worse. It clouded their judgment and they stopped using DOMAIN EXPERTISE IN ROLES [4.2.22]. Instead, they started giving the more difficult assignments to the perceived "gung-ho" crowd rather than to the people most likely to be able to do them. ... Some of the fanatics were regularly working 80-hour weeks and used the reward system as leverage to exact punishment against those who tried to work reasonable hours and balance work and family life.

The liability of providing big rewards for meeting key corporate objectives is that people who take on those responsibilities can overextend themselves, leading to personal stress and potential risk to the project. Rewarding management staff can be particularly problematic since those staff can run the risk of developing a burnout culture in their subordinates to meet development objectives [see THE OPEN/CLOSED PRINCIPLE OF TEAMS (6.1.4)].

Some factors that lead to success are difficult to measure or even identify, so it's best to orient rewards around the organization's shared value system of what is important to achieve. Scoping the concern of "organization" in this context is key to the long-term success of the enterprise; in other words, a closed group can't sustain values that are inconsistent with those of the enclosing organization or the next higher level of management.

Success makes it possible to improve the work environment infrastructure, making it a more attractive place to work. This form of long-term compensation or of recognition of success is particularly important in team settings. In one organization, we used windfall funds to buy an interactive terminal, which in that era (about 1974) was a treat for the staff. On a broader scale, you can buy an espresso machine (for which Bell Labs' computer science Research department was famous), a coffee machine, or a watercooler—or build an entire culture of food. See THE WATERCOOLER (5.1.20).

Large rewards to some individuals may still demotivate their peers, but rewarding everyone on a team basis helps remove the "personal" aspect of this problem, helps to establish the mechanism as a motivator, and provides a "postmortem soother." On the other hand, see the discussion in THE ROLE OF MANAGEMENT (6.3.7) that puts individual contributions in a broader perspective. For example, are individual successes really just team success in misleading packaging?

The grounding for this pattern is empirical. There is a strong correlation between wildly successful software projects and a lucrative reward structure. Cases include QPW and cases cited at the Risk Derivatives Conference in New York on May 6, 1994 (see [Lawler 1981]). The place of reward mechanisms is well-established in the literature [Kilmann 1984].

At the PLoP review of this pattern, Dennis DeBruler noted that most contemporary organization cultures derive from the industrial complex of the 1800s, which was patterned after the only working model available at the time: military management. (One common model of military management is reward individually punish corporately, which leads to a fear of failing and to resentment towards those who fail.) 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 [DeBruler 1996].

Paul Bramble adds, "The trick is to be discerning—sometimes it's the quiet plodders who generate the success, and you have to be able to see past the self-promoting employees to see who really gets the work done" [Bramble 2003].

See also COMPENSATE RESULTS [Beedle 1997].

4.2.26 Failed Project Wake *

The Trident project was the most exciting project I had ever worked on. We were a small team with an aggressive schedule, but we made good progress and were actually ahead of schedule. Then one day the company made a major business decision that meant that the Trident project was probably unnecessary. Sure enough, the project was canceled a few days later. We all agreed that it was probably the right decision, and we appreciated the speed with which it was made; nonetheless, the decision still hurt.

For a week, we walked around in a fog. We did nothing. Finally, we took the afternoon off and had a party at someone's home. We brought our families and played croquet in the backyard. After that, it was much easier to move on to the next project.

... projects fail for a variety of reasons. Many of these reasons are attributable to the team involved in the project; in fact, this pattern language is designed to help with many such problems. But software developers don't work in a vacuum. Many external factors contribute to the success or failure of any project. Changes in the market, for example, can doom a product before it ever gets out the door. The greatest, hardest-working team in the world still might have their project canceled in spite of their best efforts.

Canceling a project, even for the best external reasons, is particularly demoralizing to a team that has put its heart and soul into it.

It doesn't matter much that the team members fully understand the reasons behind the cancellation: Regardless, they still feel bad. They feel powerless, somewhat apathetic, and sometimes betrayed. At best, they will need some downtime, even if they have another project to jump into immediately. At worst, they may quit.

They may note that successful projects are rewarded, but it wasn't their fault that their project was canned. This feeling of inequity can be quite strong.

Therefore:

Hold a wake for the failed project (e.g., much like an Irish wake, a party for the dead).

Don't try to placate the team with false praise about the project's "success." They all know the project bombed.

Go ahead and make it a big party, that involves more than just cake and punch in the cafeteria. And make it a real party, not a project retrospective. There is a time and a place for retrospection, and this isn't it. ([Ackermann 2002] points out that it is possible to combine a party and a retrospective event, if you have a strong facilitator and if the main purpose of the gathering is to hold a wake.)

It's best to hold the wake offsite. Doing so helps people break from the old project and avoids even the appearance of a retrospective event.

It's even more helpful to hold the wake during working hours, perhaps one afternoon. Holding the wake during work hours sends a subtle "thank-you" message: everyone knows they don't get a bonus [see COMPENSATE SUCCESS (4.2.25)], but they appreciate some acknowledgment of their efforts.

Just like the death of a loved one, the death of a project causes a period of mourning. A wake helps people get through the stages of mourning.

It also serves as a bit of a catharsis. People will come out of it much more ready to succeed on the next project. It is particularly important for upper management to express explicit appreciation for the effort, especially when the failure is due to business decisions rather than to decisions made by the development team. Paul Bramble notes that such acknowledgment helps "calm people down and ... be less worried about their future at the company" [Bramble 2003].

4.2.27 Don't Interrupt an Interrupt

See section 4.1.26.

4.2.28 Developing in Pairs **

Randy and I work on a software tool together. Over the years that we have developed it, we have spent a lot of time at each other's desks. We often write code, debug, and test together. It is not uncommon for one of us to type, while the other one indicates what to type. Oftentimes, the one who is not typing will point out typographical or logic errors. Such feedback can be annoying, but it certainly reduces the cycles of compiling and debugging.

Nobody told us to work this way. We just found that it works well.

... a development organization is in place, and people have started to commit to work and are about to start building the work artifacts. Some of the work may be allocated on the basis of CODE OWNERSHIP (5.2.13). There is enough understanding about overall requirements to start work, though many requirements may have loose ends.

Some people don't want to work alone, and working alone has great risks of blindsiding and producing misfits. Furthermore, you need to take into account people who don't want to work alone, and you must engage people who are working alone but probably shouldn't be.

People sometimes feel they can solve a problem only if they have help. Some problems cannot be solved by just one person, so people who are comfortable working alone should still work closely with someone else who at least provides another set of eyes to look over the work.

It takes extra resources to put people in work pairs in real time. In fact, one might argue that code walk-throughs and inspections and reviews provide sufficient compensation for these problems. But these reviews are usually analytical rather than opportunistic. Reviews set up an adversarial context where the critics don't have the same stake in the results as the programmers. Furthermore, reviews catch problems after the programmer has committed to the corresponding structures and algorithms and has expended a lot of effort in elaborating them, rather than catching these problems at the conceptual stage. And many of these decisions are too complex to arise in design reviews or simply can't be foreseen until the programmer grapples with implementation; nonetheless, these problems can be weighty enough to threaten the viability and long-term health of the code.

Only a limited number of people can sit in front of a keyboard and screen each other's efforts. Communication and coordination efforts increase nonlinearly with the number of people. So you can't always create a team that works together as a unit to contribute to an artifact in front of a single screen.

Therefore:

Pair compatible designers. Together, they can work together to produce more than the sum of the two individually.

There are two keys to making this effort successful. First, the individuals must be able to work well together, which means that pair assignments must not be made arbitrarily. In fact, because a pair is in reality a small team, SELF-SELECTING TEAM (4.2.11) must be applied. The chief consideration for creating a pair is that the two *want* to work together.

Second, the style of pair development must not be dictated; instead, it should be left up to the individuals. Simply put, there should not be a rule that no line of code can be written unless both people are at the keyboard. Instead, give the pair the assignment and let them figure out how to do the development. Note that this practice supports FEATURE ASSIGNMENT (5.2.14).

The pair needn't always comprise developers only. In BUILD PROTOTYPES (4.1.7) and in many other activities, one of the pair can be a customer, systems engineer, or technologist representing an area of risk being explored by the prototype. At Mediagenix, for example, a tester sometimes pairs with a developer; the tester tests the code, and the developer fixes bugs. This pairing makes it possible to circumvent the project's formal bug-reporting bureaucracy, thus reducing the time to a stable load [see also COUPLING DECREASES LATENCY (5.1.22)].

Overall, this process leads to a more effective implementation process. Contrary to simplistic reasoning, experience has shown that it may cost less overall to program in pairs than to have one coder work on code at a time. In an analogous study, it was recently found that it actually saves hospitals money if a pharmacist follow doctors on their rounds as they make prescriptions. The pharmacist's insights in correcting the doctor's errors (e.g., prescribing drugs that are incompatible with each other) saved more money (in additional health care costs) than

the cost of the pharmacist. In addition, this pairing capitalized on the pharmacist's dead time between activities.

A pair of people is less likely to be blindsided than an individual developer. Also, such pairings help ensure that SOMEONE ALWAYS MAKES PROGRESS (4.1.20).

If enough people use DEVELOPING IN PAIRS, and if the pairs rotate occasionally, the emergent structure and emergent organizational behavior contribute to cross-training efforts, information sharing efforts, and trust.

Compare this pattern with GROUP VALIDATION (4.2.32) and RESPONSIBILITIES ENGAGE (5.1.14). One special case of DEVELOPING IN PAIRS occurs when one developer asks another developer (or other suitable expert) to perform a desk check of recently written code, which is much less costly and not less effective than traditional code inspections, code walk-throughs, and code reviews. Though probably less effective than the "canonical" form of DEVELOPING IN PAIRS, this pattern's worth has been validated empirically [Votta 1993].

There are other configurations that have many of the same dynamics as DEVELOPING IN PAIRS, but that do not involve the pairing of a dynamic duo. At Mediagenix, we found teams that "programmed with the projector," where the computer screen was projected onto a wall, and a team jointly commented on and guided the work as one person sat at the keyboard. At Bell Laboratories, Joe Davison, Ricky Spiece, and Martin Biernat worked on a team where one of them stood at the whiteboard and one of them sat at the terminal, "thinking out loud" and representing the customer. In this case, the code was written on the board and transcribed into Smalltalk while the third person performed a real-time code review.

And in what might be viewed as another slant on paired programming, Doug Lea used a variant of the clean-room methodology that employed a single programmer who takes on a role as programmer and then takes on a role as tester. Clean-room techniques separate these two roles to make sure the individual can focus exclusively on the task at hand. In the extreme application of the clean-room methodology, developers are not allowed to use the compiler to check their own code; instead, they must await such feedback from the tester. Doug mimicked this behavior by wearing two hats. In one sense, this process is as unlike popular pair programming as possible: Each "side" of Doug worked in isolation. But the interplay between the two perspectives is where the power lies: bringing multiple perspectives to bear on the same artifact with tight coupling of minds. Doug's mind provided the tight coupling.

See [Williams 2002].

4.2.29 Engage Quality Assurance **

FSA supervisor and farmer-client examining silage from a trench silo, Sheridan County, Kansas.

... you have a development organization mature enough that roles have been congealed and a customer has been engaged [ENGAGE CUSTOMERS (4.2.6)]. You need some filter between the two to both facilitate and regulate interactions between them.

Customer engagement is a key element of QA. Though developers may feel they get everything right, a good dose of customer reality helps bring the perspective that development of perfect software is impossible.

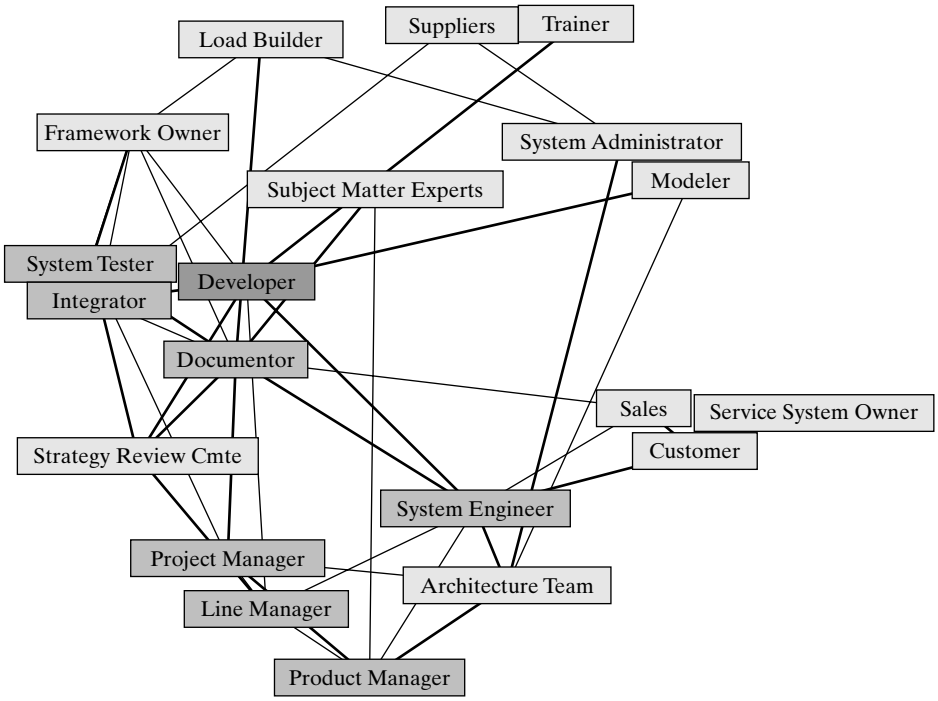
Too many organizations defer quality until "later" or equate QA with testing which occurs late in the development process. Yet, success depends on the production of high quality work, and early feedback is important to address fundamental quality problems.

It's important to perform testing, and most developers in fact perform their own testing. But individuals easily get blindsided by their own design thinking in terms of what needs to be tested. Further, they may use testing as their quality criterion. However, you can't test quality into a product; instead, you can only build a product and test its quality.

Therefore:

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

QA was central to the development of Borland's QPW:



The QA organization should be outside the context of development; in other words, the planning and reporting of tests should not be accountable to the development organization. The development organization develops a sense of accountability for delivering a quality product, since their own view of their reputation is linked to the ability to minimize the bugs that "those people in QA" find.

QA should interact with marketing in order to understand the needs and challenges a system will face.

QA people have skills and experience that allow them to view customer needs from a perspective that may not be reflected in requirements or other articulations of needs. A good example is security companies that develop security software utilities for commercial applications. Their own probing of the operating system often uncovers security holes, and then they work with the vendor to fix the problems.

Having engaged QA, the project is ready to approach the Customer. With QA and the Customer engaged, the QA process can be put in place (e.g., use cases can be gathered).

There are at least two reasons for making QA a separate organization from that holding Developers' allegiance. First, test development shouldn't be blindsided by the Developer perspective. If both the Developer and QA perform their own tests, testing becomes a double-blind experiment with the software as a subject. Second, QA should remain outside the domain of influence of the development organization in the interest of objectivity. This is an obvious pattern in QPW.

Indeed, ENGAGE QUALITY ASSURANCE requires a separate QA organization, which is in contrast to the ideals espoused in XP. XP advocates extensive unit testing, but in the words of Kent Beck, "documentation, design, formal review, separate QA; it's all a waste of our time" [Waters 2000]. This response may be a reaction to organizations that have a separate QA organization, but do not engage it. Such a setup is a recipe for disaster: You have the overhead of a separate organization without any of the benefits. In order for a separate QA organization to be effective, it must have frequent and positive interaction with development.

Note that QA should be engaged early in the project. By the time testing starts, it is too late to build the trust needed for QA to proceed smoothly, a problem that is clarified in GET INVOLVED EARLY (A.5.13) [Delano Rising 1998]. It is not just the developers' responsibility to engage the testers; instead, the testers must reach out to the developers as well (see DESIGNERS ARE OUR FRIENDS (A.5.10) [Delano 1998]).

See also APPLICATION DESIGN IS BOUNDED BY TEST DESIGN (4.2.30).

4.2.30 Application Design Is Bounded by Test Design *

An M4 tank tops the ridge on a test course. The tank was designed to meet all of the challenges of the test course, which should simulate all of the extremes of the field.

... a development organization has mechanisms to document and enforce the software architecture, and it has developers to write the code. You are planning how to engage your customer. A Testing role is being defined.

When do you design and implement test plans and scripts?

Test development takes time and cannot be started just when the coding is done. One cannot have the mindset that, "we will know what to test once we have coded it."

Scenarios are known when requirements are known, and many of these scenarios are known early in the process [see SCENARIOS DEFINE PROBLEM (4.2.8)].

Test implementation teams need to know the details of message formats, interfaces, and other architectural properties in great detail (to support test scripts and test jigs). Both software developers and testers need to work closely together from the same "scripts"—the use cases that define customer needs.

Yet, external tests usually do not reflect an understanding of the internal software structure, so much test development can take place in parallel with design and implementation of the deliverable software. Implementation changes daily; as such, there should be no need for test designs to track ephemeral changes in software implementation.

Therefore:

Use-case-driven test design starts when the customer first agrees to use case requirements. Test design evolves along with software design, but only in response to changes in customer use cases; the source software, thus, is inaccessible to the tester. When development decides that architectural interfaces have stabilized, low-level test design and implementation can proceed.

Software designers can and should use test specifications as a major touchstone for requirements.

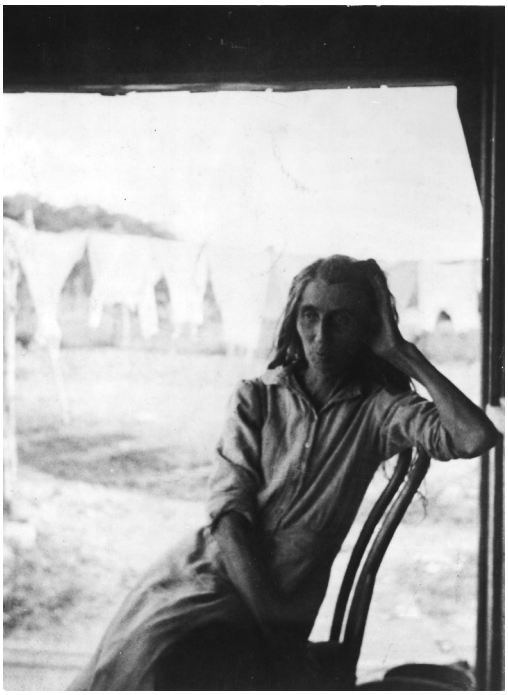
This pattern provides a context for SCENARIOS DEFINE PROBLEM (4.2.8) and complements ENGAGE QUALITY ASSURANCE (4.2.29). Once the expectations are established between the testers and developers in the context of customer expectations [perhaps through FIREWALLS (4.2.9) and GATEKEEPER (4.2.10)] you can approach the customer to capture use cases.

Making the software accessible to testers causes them to see the developer's point of view rather than the customer's point of view and leads to the chance that they may test the wrong things or test 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.

4.2.31 Mercenary Analyst

See Section 4.1.24.



4.2.32 Group Validation *

Testing a homemade screen door for strength.

... an activity such as analysis, design, or implementation has been completed and is about to be assessed.

Product quality is crucial to the success of the enterprise. QA usually assesses the quality of the end product, performing only black-box validation and verification. Development groups may bring many insights on product problems and opportunities. Individuals, however, may not have the insight necessary to discover the bug plaguing the system (there may be issues with lack of objectivity).

Therefore:

Even before engaging QA, the development team—including the customer—can validate the design. Techniques such as CRC cards and group debugging help socialize and solve problems. Members of a validation team can also work with QA to fix root causes attributable to common classes of software faults.

The software shouldn't be the only focus of debugging and review. Recurring types of software bugs may point to systemic problems in the structure of the organization itself. For example, if the project is seeing a high rate of mismatches in interfaces between components, the integration may be taking place too quickly to allow all of the team members to keep in step with the current state of the architecture. The organization can write new patterns to solve these systemic problems [UPDATING THE PATTERNS (3.3)]. See [Fagan 1976].

One can create a culture where the quality of the system is constantly brought into focus before the whole team. Problems will be resolved sooner than if they are deferred to the "official" QA function, which typically interacts with the project at the boundaries of design and coding. The cost of this pattern is the time expended in group design/code debugging sessions.

The CRC design technique has been found to be a great team-building tool and an ideal way to socialize designs. Studies of projects inside AT&T 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 the customer and the developer, using the patterns mentioned later in this pattern.

There is an empirical research foundation for this pattern. An article in the Communications of the ACM (CACM) [CACM 1979] shows that team debugging contributes to team learning and effectiveness. A contrary position can be found in [Meyers 1978], though this study was limited to fault detection rates and did not evaluate the advantages of team learning.

There are times when reviews do not need not be a group effort; sometimes, all it takes is a little help from a friend. DEVELOPING IN PAIRS (4.2.28) is one example, and the kind of desk checks mentioned in [Votta 1993], where one person liberally marks up the work of another, also can be effective (Votta shows that this mode of review is almost as effective and much less costly than a meeting). The CREATOR-REVIEWER (A.5.8) pattern [Weir 1998] calls this person-to-person a "distribution review" as opposed to a "meeting review." Doug Lea once took this approach to an extreme, working on a one-person clean-room programming team where he played the roles both of programmer and reviewer, with no use of a compiler to validate the code between the steps (see DEVELOPING IN PAIRS). We imagine the psychological forces must have been both interesting and compelling.

STAND-UP MEETING (5.2.7) is an informal form of this pattern.