Patterns of Project Management Risk Reduction

Alistair Cockburn, arc@acm.org

Risk reduction would be the favorite pattern among project managers, if only we could figure out how to describe it. I put forward a format for capturing risk reduction patterns, with six less obvious patterns and a table indexing situations into documented patterns.

Each pattern is written as though it were a medical diagnosis:

- It starts from subjective information ("indications"): how you feel, complaints you hear in the hall.
- It lists the forces that you are trying to balance, how you are trying to balance them.
- It gives a sample recommended action.
- It documents overdose: what difficulty you will encounter if you over-apply the recommendation.
- The next problem you might need to solve is given in the resulting context.
- There is a section for the principles involved. These are included for future researchers to examine, to form more solid theories of project management.
- It includes a typical situation you may find yourself in, or a previous project found itself in.

A strict application of the medical analogy calls for a "validation procedure", tests to run to confirm the diagnosis. One could hope to eventually develop such procedures, but we are not there yet.

Each situation results from some force getting out of balance. So, one pattern may remedy several situations. Similarly, any one situation has several possible remedies, depending on small variations in the situation, or your personal management style.

This set of patterns is intended to grow. A large number of patterns or prescriptions needs to be organized. I have divided the patterns by primary problem - quality, productivity, efficiency, etc. A project manager would browse the section on distractions, rather than working only through the index.

The patterns have been commended from both object-oriented and traditional projects.

Here, now are thumbnail sketches of the six included patterns. Next come the six patterns in detail. At the end is a table that characterizes known risk reduction patterns by problem and symptom.

1. Ownership: Owner per Deliverable
Sometimes many people are working on it, sometimes nobody. So...
Make sure every deliverable has exactly one owner.
(General pattern with specializations:)
Function Owners / Component Owners
Team per Task

2. Ownership: Function Owners / Component Owners
If you organize teams by components, functions suffer, and vice versa. So...
Make sure every function has an owner, every component has an owner.

3. Distractions: Someone Always Makes Progress
Distractions constantly interrupt your team's progress. So...
Whatever happens, ensure someone keeps moving toward your primary goal.
(General pattern with specializations:)
Team per Task
Sacrifice One Person
Day Care

4. Distractions: Team per Task
A big diversion hits your team. So...
Let a subteam handle the diversion, the main team keeps going.

5. Distractions: Sacrifice One Person
A smaller diversion hits your team. So...
Assign just one person to it until it gets handled.

6. Training: Day Care
Your experts are spending all their time mentoring novices. So...
Put one expert in charge of all the novices, let the others develop the system.

(1) OWNERSHIP: OWNER PER DELIVERABLE

Thumbnail:

Sometimes many people are working on it, sometimes nobody. So...

Make sure every deliverable has exactly one owner.

Indications:

You detect a "common area" (an area where multiple people are updating concurrently, without dominant ownership), or an an "orphan area" (an area where no one works or accepts responsibility).

You hear: "What happens when two people need to program the same function?"

Classes are beginning to look like the refrigerator in a shared apartment.

No one is updating the class diagram.

You have multiple teams working on one task, or one person working on many tasks.

Forces being balanced:

You want people to share. You want consistency.

You want every deliverable to have internal integrity, to be consistent and maintained.

…but…

When people share incompletely, you get a common area.

If no one is assigned responsibility, you may get ownership, or a common or orphan area.

Recommended action:

Make someone responsible for each deliverable: project task, project end results, overall consistency. Ask and make sure you can answer, who is answerable for each of these things:

- overall system architecture / application software architecture
- user interface design quality / OO design quality / code quality
- requirements / domain model / each class / documentation/test cases.

Resulting context:

Ownership needs can conflict. You may have to set up a conflict management procedure.

Overdose effect:

The "Swiss accountability" effect: Ownership of every thing on the project can be perceived by some people as wonderful, and others as a nuisance.

Conflict management dominates the team's energy (indicates improper ownership boundaries).

Related patterns:

Function / Component Owners - addresses the particular ownership functions and components.

Team per Task - where a task is the unit of ownership and conflict.

Day Care - addresses training as a distinct deliverable.

Principles involved:

Sometimes a task is so onerous that only by making it part of a person's job responsibility can you ensure that it gets done. This is the case in updating class diagrams and creating documentation, often-found orphan areas.

People cannot track other people's intentions, making a common area hard to clean up. It is also not in any one person's local interests to do so (see Senge 95).

Sample situations:

A. Object-orientation is itself an example of *Owner per Deliverable.* A software module owns all of the resources, data and computation, it needs around a particular purpose. This organization of software reduces the trajectory of change for the software.

B. I was asked on a project visit, "What happens when two people try to program the same function?" That question implies either that the functioning of the system has not been adequately partitioned, or two people are putting their fingers into the same class. See *Function Owner / Component Owner*. It is not surprising to discover people redundantly designing the same function under those circumstances.

C. *Orphan area.* The class diagram often is out of date, with it being no one's job to update it. Find out who owns that segment of the class structure, and assign someone to own the diagram, as part of their job responsibility. In order not to waste too much of that person's time, decide on the smallest set of moments when the class diagram needs to be current. Note that every piece of final documentation is a deliverable to receive ownership.

D. There is no one person responsible for the system architecture or some other major element of the system. I repeatedly find that there is no one person who answers for the quality and currency of the class diagram; no one person who answers for the quality an consistency of the user interface, the program code, the performance of the system.

Reading:

Senge, P. et al, The Fifth Discipline, Currency Doubleday,1990 discussed the effects of common areas in the context of feedback loops.

(2) OWNERSHIP: FUNCTION OWNERS / COMPONENT OWNERS

Thumbnail:

If you organize teams by components, functions suffer, and vice versa. So...

Make sure every function has an owner, every component has an owner.

Indications:

Your teams are organized by function or use case, with no component ownership.

Your teams are organized by class or component with no function or use case ownership.

You get the question: "What happens when two people need to program the same function?"

Forces being balanced:

You want ownership and consistency in the functions.

You want ownership and consistency in the components.

You want components to be shared across teams.

…but…

Ownership by function turns components into shared areas.

Ownership by components turns functions into orphan areas.

Recommended action:

Make sure every components has a responsible owner and make sure every delivered function has a responsible owner. The component owner answers for the integrity and quality of the component. The function owner ensures that the function gets delivered. If the component owners all refuse to incorporate something needed to deliver end functionality, the function owner sees that the missing code is put in a function-specific place.

Resulting context:

Possible friction between component and function owners. The need for the Envy/Developer model of ownership, in which there is a common part to each component or class, and an application-specific part.

No conflict resolution is needed in this pattern, since the component owner has right of refusal to any request, and the function owner has recourse to another to get the job done.

Overdose effect:

None known, since there is no overdoing the pattern.

Related patterns:

Owner per Deliverable - the general form of this pattern.

Team per Task - where a task is the unit of ownership and conflict.

Principles involved:

Common area, orphan area, as discussed in *Owner per Deliverable.*

Sample situations:

A. It is frequent that a project starts out with teams centered around classes or components. At delivery time, the end function does not work. Each team says, "I thought you were taking care of that. It doesn't belong in *my* class."

On one such project, the project manager assigned each function to a responsible owner. That person had to negotiate between the teams to see if some team was willing to pick up the gap, or to see that some extra, special code was created to cover the gaps between the classes.

B. It is also common that project start with the alternate scheme: teams centered around use cases. They soon find that, with nobody responsible for cleaning up a class, the class develops into an arbitrary collection of state variables and functions. At that point, nobody *can* clean it up. In one team of three I visited, the entire team sat around one workstation one day to clean up their classes. All three people were needed to understand the code.

C. *Brooklyn Union Gas* and a few other projects used both class and function owners. A class owner might refuse to put a requested function into a class. They would say it was not part of the class's responsibility, only a special need for an isolated function. The function might end up in the special class for the use case.

Fortunately, there are tools such as Envy/ Developer that allow a method/functions to be attached to a class just for a specific application. They support creating ownership around common parts of a class, and application-specific parts of a class.

D. *Envy/Developer.* The Smalltalk tool Envy/Developer directly supports this model. The presence of that tool and this pattern changes the economics of the argument that view classes are needed separately from model objects. Envy/Developer makes it straightforward, safe and local for one application team to add their function-specific methods to the class.

Reading:

none.

(3) DISTRACTIONS: SOMEONE ALWAYS MAKES PROGRESS

Thumbnail:

Distractions constantly interrupt your team's progress. So...

Whatever happens, ensure someone keeps moving toward your primary goal.

Indications:

Non-primary tasks are dominating the team's time, keeping it from moving forward with their primary goal.

Common complaints of distraction.

Forces being balanced:

Need to pay attention to every task, including small diverting ones.

Need to complete the primary task by an important date.

…but…

<This section is filled in within the specializations of the pattern.>

Recommended action:

Whatever you try, ensure that someone on the team is making progress on the primary task.

Resulting context:

Various, depending on the tactic employed. You will, however, be closer to your final goal, which is not always the case when dealing with distractions.

Overdose effect:

You may eventually get into trouble for not adequately addressing the distractions. If you have too many distractions, you may have a symptom of some other problem.

Related patterns:

Specializations:

Team per Task - separate tasks into sympathetic sets.

Sacrifice One Person - assign only one person to the distraction.

Day Care - separate the task of training task from that of producing software.

Principles involved:

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

Sample situations:

A. *Scylla and Charybdis.* In the ancient Greek story, Odysseus had to get his ship past Scylla and Charybdis. Scylla was a six-headed monster guarantees to eat six crew members, but the rest would survive. Charydbis was a whirlpool guaranteed to destroy the entire ship. In this paradigm of the dilemma, Odysseus chose to sacrifice six people so that the rest would get past Scylla's cave.

B. *Atalanta.* In the Greek story, Atalanta was assured by the gods that she would be the fastest runner as long as she remained a virgin. So she told her father, the king that she would only marry the man who could beat her in a foot race. The losers were to be killed for wasting her time. The successful young man was aided by a god, who gave him 3 golden apples. Each time Atalanta pulled ahead, he tossed an apple in front of her. While she paused to pick up the golden apple, he raced ahead, and eventually won.

You could interpret this story as containing the moral that Atalanta should not have stopped to pick up the apples - that would also illustrate the point of this pattern. I choose to view it more metaphorically, that 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.

B. Other examples are given in the specializations.

Reading:

Csikszentmihalyi, M., Flow: The Psychology of Optimal Experience, Harper Perennial, 1990.
DeMarco, T., Lister, T., Peopleware, 1976.

(4) DISTRACTIONS: TEAM PER TASK

Thumbnail:

A big diversion hits your team. So...

Let a subteam handle the diversion, the main team keeps going.

Indications:

"We have too many tasks, causing us to lose precious design cycles."

"We are getting distracted from our primary purpose."

There is a common complaint that the group is getting too many distractions (marketing requests, management requests) from the outside.

Novices needing training overwhelm the experts' ability to make progress.

Requirements gathering is taking longer than the schedule can allow.

The schedule needs major, immediate attention.

The version in test needs attention, but so does the version in development.

Forces being balanced:

We need to pay attention to every task, including small diverting ones.

We need to complete the primary task by an important date.

We want people to be satisfied with their jobs.

…but…

It takes significant time for people to switch between tasks.

There is such a thing as the "primary" task.

Recommended action:

Split the team into two. Sort the activities so that each team has one primary task with additional, sympathetic activities. Sitting in meetings, answering phone calls, writing reports, for example, are non-sympathetic to designing software. Arrange it so that each team can focus on its primary task, and each task has a dedicated team member.

Resulting context:

A smaller primary team with a secondary team working on the interrupting task.

Overdose effect:

It is not worth splitting up the team's task set because the working synergy between people that is lost is more harmful than the dedicated time gained per task. You eventually get one-person teams.

Related patterns:

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

Owner per Deliverable - the general form of ownership and accountability.

Function / Component Owners - team for each artifact, as well as the task of designing it.

Someone Always Makes Progress - the general distraction management pattern.

Sacrifice One Person - specialization to lose only one person.

Day Care - addresses training as a separate deliverable from the software.

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 90, DeMarco 76). It is when the 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, it is 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 flow, the designer must have time to make actually progress, which may be another 10 minutes. Any significant interruption within the half hour essentially causes the entire half hour to be lost. As it takes energy to get into the flow, 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, 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 to group tasks into sympathetic sets. Requirements and analysis involve meetings, reading, and writing. Design and programming require concentration on the implementation technology and keeping a great number of details in the head.

Parallelism vs. time-slicing. Time-slicing can be more attractive in terms of job satisfaction - each person will do design some part of the time. The significant time 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*, *Day Care*, and Coplien's *Firewalls* for examples).

Sample situations:

A. Concurrently gathering requirements and designing software.

On one project we tried having each person do requirements, analysis, design and programming. We thought the developers would enjoy the change of activity, that this would reduce the meetings and bureaucratic documentation exchanged between people.

What happened was that the first two activities were so different from the latter two that people were unable to switch easily between them. After having attended and documented meetings for much of the day, it was difficult to start working on the design and programming. Here, as with

changing between bug-fixing old code and writing new code, every time a designer was pulled away from her or his work, it cost an additional hour to recover their train of thought.

We applied Team Per Task, and split the teams along task lines. Requirements gathering and analysis went with designated people in each team, and design and programming went with the others. The result was that the requirements/analysis people sat in meetings, read and wrote specs, examined interfaces and the like. They communicated their findings to the designer/programmers - orally, for the most part, since they were closely linked on the same team (we used *Holistic Diversity*[1]). The designer/programmers stayed in their train of thought, getting fresh input from their requirements colleagues. Some of the people put onto requirements really wanted to program, so this was quite a sacrifice for them (*Sacrifice One Person*).

Two things we did not do. We did not put the requirements/analysis people into a separate team (*Holistic Diversity* again). A team was jointly responsible for a section of the system, from requirements to delivery. The splitting was within each team. We also did not require the requirements group to document their decisions for the designers benefit (they did document for the project's benefit). The requirements and design people were in close contact at all times, and most information passed orally. There was, therefore, no "throw it over the wall" effect. These were both important teaming decisions made earlier, which we were intent on preserving.

B. Training distracts the experts. See *Day Care*.

C. Other examples are found under *Sacrifice One Person*.

Reading:

Csikszentmihalyi, M., Flow: The Psychology of Optimal Experience, Harper Perennial, 1990.
DeMarco, T., Lister, T., Peopleware, 1976.

[1] *Holistic Diversity*: Development of a subsystem needs many skills, but people specialize. So… Create a single team from multiple specialties. From Cockburn 96 (see master table at end).

(5) DISTRACTIONS: SACRIFICE ONE PERSON

Thumbnail:

A smaller diversion hits your team. So...

Assign just one person to it until it gets handled.

Indications:

As for *Team per Task*, except the distraction is smaller; it can be handled by one person.

Forces being balanced:

We need to pay attention to every task, including small diverting ones.

We need to complete the primary task by an important date.

We want people to be satisfied with their jobs.

…but…

It takes significant time for people to switch between tasks.

There is such a thing as the "primary" task.

The diverting task seems to be small, but is very important.

Recommended action:

Assign one person full-time to the distraction rather than several people part time.

Resulting context:

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. Before trying to make the sacrifice part-time work, evaluate the loss of "flow" time that will result from the person dealing with both this distraction and some other task.

Overdose effect:

If this keeps happening, you will have no one performing the primary task, and you ought to examine why you have so many distractions in the first place.

Related patterns:

Owner per Deliverable, Someone Always Makes Progress, Team per Task - the more general forms.

Day Care - training is the distraction. Produces mentor as a profession (see Principles).

Firewall (Coplien 94) - the distractions come from outside the team, so one of the developers is sacrificed to act as project manager. Produces project manager as a profession.

Mercenary Analyst (Coplien 94) - documentation is the distraction, and someone is hired to take care of it. Produces technical writer as a profession.

Gatekeeper (Coplien 94) - the constant inflow of technical information is the distraction, and one person is assigned managing that information as a distinct, part-time task.

Principles involved:

Same as for *Team per Task*. It looks like handling the distraction should be a part-time job, but this just illustrates the significance of the time spent getting into mental flow.

Parallelism, sacrifice, or profession? If the people do not like the task, they consider it a sacrifice. If they like the task, it becomes their profession. Thus, *Firewall* gives rise to the profession of project management, *Day Care* gives rise to the profession of mentor.

Sample situations:

A. *Project schedule.* The schedule was out of date. We thought it would be fair to let each person predict their own work. That would spread the experience, discomfort and load. What happened was that progress came to a 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 head.

One team used *Sacrifice One Person*. The person who drew the short straw did the whole team's estimation. The others got on with their work. That team continued to move forward while the other teams were at a standstill. The person working on the schedule really felt sacrificed, very much as in the story of Scylla and Charybdis.

B. *Simultaneous development of the next release with release to QA.* One increment was entering test at the same time design was starting on the next. We thought the bug fixes would take a relatively small amount of time, and so assigned the whole team to both fixing bugs and doing new design. However, each fix broke a designer's train of thought for a period of time on the order of an hour beyond the fix. Three or four caused the designer to lose most of the day. The designers gave up on the new release, as they knew the next bug fix would arrive before they progressed on the new work.

Applying *Sacrifice One Person*, we assigned one person to bug fixes. 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 test.

Reading:

Coplien, J., "A development process generative pattern language", Pattern Languages of Program Design, Coplien, Schmidt, eds., Addison Wesley, 1994.

(6) TRAINING: DAY CARE

Other names:

"Progress Team / Training Team"

Thumbnail:

Your experts are spending all their time mentoring novices. So...

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

Indications:

"We are wasting our experts."

"A few experts could do the whole project faster."

The experts are not proceeding at the rate you or they would expect.

Training is draining their energy, time, concentration.

You have to add a batch of new people to an existing project.

Forces being balanced:

You need people trained.

You need the system built.

…but…

Trained people and delivered system are both required results of your project.

You want the newcomers near an expert, to learn what the expert knows.

Novices distract and drain experts.

Recommended action:

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 novices. Let the progress team design 85-95% of the system, let the training team focus on quality training, delivering only 5-15% part of the system. Transfer people to the progress team as they become able to contribute meaningfully.

The training team does not simply do training exercises, but actually contributes to the final system in an ever-increasing way.

Resulting context:

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 may cause conflict.

Overdose effect:

You eventually have too few people to constitute a progress team.

Related patterns:

This pattern is a cross-specialization of several given in this chapter: *Owner per Deliverable, Someone Always Makes Progress, Team per Task, Sacrifice One Person.*

Principles involved:

The principles are synergy vs. distraction, the synergy of having a novice learn directly from an expert vs. the distraction to the expert. Experts having to answer novice questions are reduced to a fraction of their productivity, without particularly raising the productivity of the newcomers. Adding one novice to an expert may cut the expert's productivity in half, adding two may cut it to a third, adding three may prevent all productivity altogether.

Assume there are X experts who work at productivity 1 each. If they could work together, they would have a total productivity of (X). Assume there are N novices who work at n productivity each, with n much smaller than 1, on the order of $1/10$.

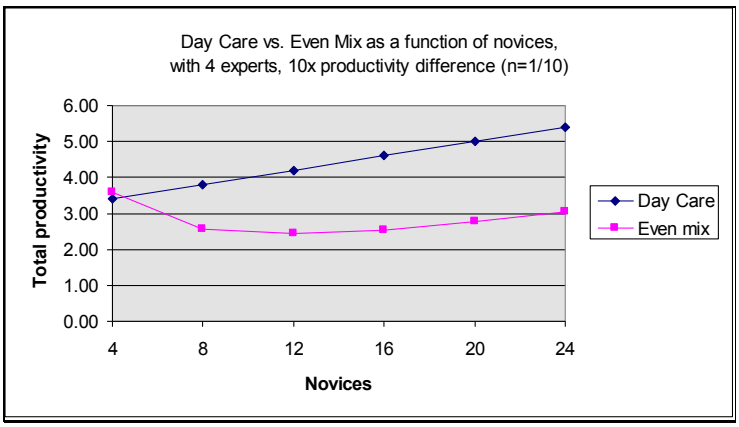
If one expert is sacrificed to train the novices, that person has zero productivity (except training novices), so the group's total productivity is

$(X-1) + N*n$ for *Day Care.*

If they 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

$(X*X/(N+1)) + N*n$ for Even Mix

The graph below shows how Day Care quickly improves over Even Mix.



The nature of the training does not matter. Design and teaching are antagonistic tasks (as described in *Task per Team*), and 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*. *Someone Always Makes Progress* protects the delivery of running software.

Sample situations:

A. Mentoring. A standard recommendation is to put 1-5 novices under each trained expert. The consequence is that the experts spend the prime part of their energies training, halfheartedly. Besides being drained of energy for designing the system, the experts typically do not have the personality, background or inclination to actually teach the novices how to do design. They are caught between trying to get the maximum out of their trainees and trying to do the maximum development themselves. Thus, they neither develop the system, nor train the novices adequately.

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

B. Adding staff. Fred Brooks, in The Mythical Man-Month, talks about the training costs of adding people to a project. These new people drain productivity from the experts. The same suggestion applies: put the newcomers in a separate team to learn the system. Move them to the progress team as soon as they are up to speed.

Reading:

Brooks, F., The Mythical Man-Month, Addison-Wesley, 1995.

Lave, Situation Learning: Legitimate Peripheral Participation, Cambridge Press, 1991, describes the use of this sort of arrangement in apprentice situations.

Diagnosis Table

Patterns appearing in this paper are underlined.

General Problem	Indications	Observations	Pattern(s)
Communication	"Our teams are not communicating." "We are doing 'throw it over the wall' development." "Our process too bureaucratic."	Teams are structured by specialty or phase deliverables, passing work to each other by written deliverables, not able to get their discoveries incorporated into connecting teams' work patterns. Hostility or lack of respect across teams.	Holistic diversity **(Cockburn 96)** Diversity of membership **(Harrison 95)** Feature Teams **(McCarthy 95)**
Communication	"I am doing everything"	No specialization. Each person assigned to do everything, resulting in waste from changing gears.	Holistic diversity **(Cockburn 96)**
Communication	"We are not united in purpose in our technical effort."	People pulling in different technical directions.	Unity of purpose (**Harrison 95)** Architect controls product **(Coplien 94)**
Communication	"We have cliques and splinter groups."	Communication distances too large, lack of a core group.	Buffalo mountain **(Coplien 94)**
Distractions	Team getting distracted. Distractions causing loss of design cycles	Non-primary tasks dominating time, keeping team from moving forward with their primary goal. Possibly, spending too much energy switching contexts.	Someone-Always-Makes-Progress Team per task
Distractions	"This is a distraction to our primary purpose." "This distraction is causing loss of design cycles."	Some important interruption taking time from all the team members. Interruption can be handled by one person	Someone-Always-Makes-Progress Sacrifice One Person Day Care Mercenary analyst **(Coplien 94)** Firewalls **(Coplien 94)** Gatekeeper **(Coplien 94)**
Documentation	"We have too many deliverables to maintain properly."	Too many deliverables make for a heavy bureaucratic load. Deliverables cannot be reduced.	Owner per deliverable
Efficiency	"We don't have time to wait for the upstream teams to get done!"	Team members sitting idle since upstream tasks not completed. Holistic diversity already in place.	Gold rush **(Cockburn 96)**
Ownership	"No one seems to own this deliverable." "No one seems to work on it."	*"Ignored area"* - nobody working on maintaining some deliverable. Ownership by classes only, making functionaliy an ignored area. Documentation and test cases easily become ignored areas.	Owner per deliverable Function Owners / - Component Owners Code ownership **(Coplien 94)** Mercenary analyst **(Coplien 94)**
Ownership	"No one seems to own this deliverable." "Several people work on it." "Who really owns it?"	*"Common area"* - multiple people working on it, nobody knows what can be discarded. The area cannot be cleaned up. Ownership by function only, making the classes	Owner per deliverable Function Owners / - Component Owners Code ownership **(Coplien 94)**

		common areas.	
Training	Experts are being diverted teaching. We are losing precious expertise.	Experts not proceeding at the rate they would expect. The training is draining their energy, time, concentration.	Day Care

References for the patterns in the table:

Cockburn, A., "The interaction of social issues and software architecture," Communications of the ACM, 39(10), Oct. 1996, pp. 40-46.

Cockburn, A., "A medical model of project management" , Pattern Languages of Program Design Conference 3, 1996.

Coplien, J., "A development process generative pattern language",Pattern Languages of Program Design 1, Coplien, Schmidt, eds., Addison Wesley, 1994.

Harrison, N, "Organizational Patterns for Teams", Pattern Languages of Program Design 2, Vlissides, Coplien, Kerth, eds., Addison Wesley, 1995, pp. 345-352.

McCarthy, J., Dynamics of Software Development, #7: Use Feature Teams, Microsoft Press, 1995.