# Architecture and Organization: Structure, Problems, and Solutions

Klaus Marquardt, Käthe-Kollwitz-Weg 14, 23558 Lübeck, Germany

Email: marquardt@acm.org or pattern@kmarquardt.de

Copyright © 2002 by Klaus Marquardt. Permission granted for the purpose of EuroPLoP 2002

Large software systems and projects develop a high degree of internal complexity by their sheer size, and are bound to experience trouble. This complexity is not limited to software design, but is also inherent to the process and the organization. Experienced software architects have learned to control complexity and to detect related problems before they have become critical. Their toolset consists of a variety of different measures applicable to the situation at hand. This knowledge of early symptom recognition and selection of a reaction can be collected in the form of diagnoses and therapies.

The presented patterns describe organizational structures and problems focusing on their diagnoses, causes and therapies. POLITICAL MONOPOLE describes an organization preparing for company-wide reuse, while FORGOTTEN APPLICATION (also known as TECHNOLOGITIS) takes a detailed look at the structure within a single project. The process and organization of integration is the topic of UGLY INTEGRATION.

## Introduction

There is no commonly accepted definition of the profession of a software architect yet. Most approaches focus on the initial up-front activities needed for large projects, more recent publications identified that architects also influence regular releases. Nowadays, agile development processes suggest that most of the architecture is defined while the system is already under development.

In a lot of projects, the architect is the only person who can view and understand the system from different perspectives, and find a common language with developers, managers and other stakeholders. He is in a position both technical and political, and is able to detect technical and structural deficiencies and address their impact on the project.

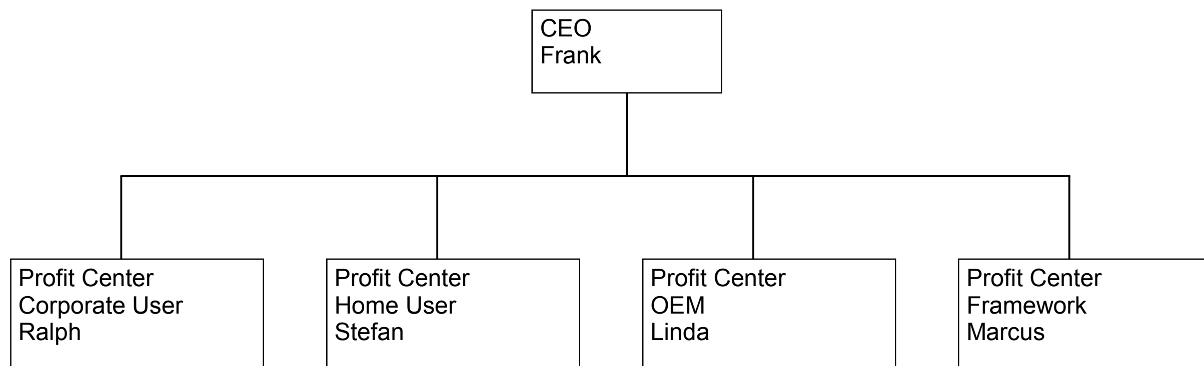
Complementing architecture with a different metaphor offers new perspectives both for agile development and for projects in crisis. The software architect is encouraged to take a role similar to a medical doctor. He examines the system and makes a diagnosis, identifies the underlying causes and starts with treatment. To the medical architect, a catalogue of diagnoses and examination techniques can be a great help to spot arising problems and give hints to effective countermeasures.

## The Medical Architecture Project

The presented patterns are part of a larger effort to make problems and solutions accessible in a medical form, as symptoms, diagnoses, and therapies. The diagnoses and their respective pathogens play a central role, as they combine different therapies and show their relations and interdependencies. The therapies are patterns or practices themselves. More diagnoses and an introduction into this approach are available in a previous pattern publication [*Marquardt01*].

## Diagnosis: Political Monopole

*A company hosts a number of business units or profit centers. One of them creates software for potential reuse within the companies' product portfolio; the other profit centers are expected to use it.*




An organization consists of business units set up as profit centers that sell products with a large set of commonalities. For the purpose of common software reuse among the present business units, an additional business unit becomes established. The expected outcome is that all products become cheaper and/or faster in development due to large scale reuse of software, and that the software itself would become profitable because its users pay for its worth.

To support the reuse effort efficiently, the company policy expects the business units to base their product development on the common software base to be developed. This makes one profit center more equal than the others. First, it has only internal customers. Secondly, it is perceived in rather different and inconsistent ways. Top management sees proof that the company-wide reuse initiative is a success; other business units frequently claim their business would go smoother without it.

You are likely to notice a tension between the business units on different levels. Managers blame each other to spend money without going anywhere, and that their business goals are not met due to things outside their influence. Employees from that more equal unit tend to exhibit a helping attitude, combined with the barely veiled knowledge that they form the intellectual and technological lead. Hardly surprising, other employees are bothered by this attitude. Communication is avoided and becomes political quickly.

**Symptoms checklist**

- ☑ A reuse effort is started from scratch
- ☑ The effort is embedded in the organization in the same way the other business is
- ☑ It has only in-house clients
- ☑ Reuse has been initiated by upper management, not by potential users
- ☑ The cross-cut business unit is expected to be profitable
- ☑ Product oriented units remain fully responsible for their profits
- ☑ The different units struggle with diverting goals, and eventually fight for expertised staff
- ☑ You sense little communication among technical staff from the different business units

**Diagnosis**: POLITICAL MONOPOLE, also known as ENFORCED CLIENTS

The company has just started a development effort that is not accepted by its future customers. They are being forced to use it against their will, and also against the criteria their success is measured with.

---

The causative germ is a business model that does not fit with reuse, combined with the perceived need to forcefully initiate reuse. However, this misfit is well motivated and not artificial. In an organization where each subunit is measured by profitability, a reuse effort is naturally put into an autonomous unit that is also expected to make monetary profit. Likewise, in an organization with technical competence and a sufficient variety of products is evenly natural to try to establish a reuse model.

There is a pathogen behind this that is hard to discover. Why would upper management be so eager to establish reuse? This is an unhealthy, unprioritized mixture of the long term goal to reuse, and the short term goal to have all decided strategies implemented by today while keeping the business running. Such impatience is often useful to establish changes, however it creates equally strong back forces that in this case counterdict the intention.

---

The architects' ability to cure an organization is limited. Architects have no control on the business model, just as the people within their direct influence sphere. Additionally, it does not make sense to change the structure, as no organizational structure is obviously superior to the outlined one. Neither should the providing software unit control any of those who create profitable products, nor can one (or several) of these control the software unit.

However, the following therapies have the potential to help a profitable organization to begin with serious reuse and to soothe some of the negative implications of unchangeable decisions. While none of them is actually curative, they diminish the gap between the different business models.

To understand how any such therapy can be effective nonetheless, a course or reading in systems thinking and change management is highly recommended. This is like a meta therapy helping you to invent your own medications for the specific situation at hand. Systems thinking enables you to understand the actions and reasons behind, and to react and proact accordingly. A basic knowledge is sufficient in most cases, you do not need to become an expert. Suggested readings are the works by Senge [*Senge90*, *Senge99*].

Successful reuse efforts can be initiated at the grass root level among engineers, with relatively little, though supportive management involvement. Apply LET REUSE GROW to start reuse in your organization without overly ambitious expectations. A good start might be to initiate an OPEN SOURCE DEVELOPMENT project.

On the process side, all reuse efforts will profit from an implementation of REWARD THE ADOPTER to become commonly accepted and thus be successful. INFORMAL SUBDUCTION helps to balance the control over reused elements between creators and adopters. It is most reasonable when you clearly distinguish between initial and SUBSEQUENT CUSTOMERS.

Two further therapies can be applied when you have influence on the company level. COMPONENT BASED DEVELOPMENT is a style of technical architecture that can foster cooperative development and become profitable mid-term. To establish a company wide business model requires more effort for which DEFER REUSE PROFITS is a mandatory precondition that you should communicate explicitly.

**Therapy overview**

| | Applicability | Effect | Related therapies |
|---|---|---|---|
| LET REUSE GROW | Preferably before changes to the organization take place | Preventive | |
| OPEN SOURCE DEVELOPMENT | Any time during the project. Also after project completion, and at your next project | Palliative, remission possible | |
| REWARD THE ADOPTER | Always | Supportive | |
| INFORMAL SUBDUCTION | Early in the reuse effort | Supportive | Might be a specific way to REWARD THE ADOPTER |
| SUBSEQUENT CUSTOMERS | Early in the reuse effort | Supportive | Supports INFORMAL SUBDUCTION |
| COMPONENT BASED DEVELOPMENT | Early in the project | Palliative, remission possible | |
| DEFER REUSE PROFITS | Early in the project, for an estimated time | Palliative | |

## Let Reuse Grow

Successful reuse comes after the first use. Software is build for a first use before you can consider reusing it. After it has proven successful, a second and third project might consider it reuse-worth and take the existing design and code. By now, you need technical means to facilitate reuse and ensure the software's reusability (for more details see e.g. PREPARE REUSE [*Marquardt01*]).

Reuse efforts from top management are doomed unless the organization is radically changed. However, management can encourage and facilitate reuse by rewarding small steps towards a company-wide vision. The key to successful reuse is to gather experience, both technical and managerial, before making a business case – even on virtual money.

LET REUSE GROW can be successful when it comes at a cost invisible to top management, and is limited to developers and architects. Alternatively, it might be fostered by top management using REWARD THE ADOPTER. Depending on your situation, a number of different side effects can occur, including a high turn-around rate of developers within and outside the company. Two different overdose effects have been observed. The reuse visibility never reached to the companies' level, leaving the impression in upper management that the development was incapable to reuse. Another overdose effect is burn-out syndrome of those individuals who take this personal effort to heart and never get acknowledgement for it.

> *A large company embedded similar software in a number of products. To increase profit, a new business unit (BU) was founded with the task to deliver a framework for a family of products. The other BU's act at a consumer market, while the Framework PC has internal customers only. To ensure reuse, the BU's are forced to use the framework and to pay a predefined price for it – even before anything like a framework is ready. After the framework project failed and the BU was extinguished, the team and the code base became absorbed by another BU for a project that was initially planned as the first framework customer. This time all political and technical framework aspects were deferred until the product was shipped. For a series of similar products, the product software was then re-evaluated and parts extracted for reuse. This loop took about four years.*

## Open Source Development

When you know about a software area that is interesting to several projects or developers within your company, you might propose a joint effort organized as an open source development. Open source development means that any developer may contribute or use, and that a small group of people control the integration and release. This introduces an alternative business model for reuse in which the most interested project can make the most relevant contributions. The companies' open source depository can serve as a basic germ for software reuse.

One fundamental mechanism of open source development is pride. Make your excellent (or cool, whichever is best) code available to your peers and receive acknowledgement in turn. This translates into a gift culture [*Gabriel00*] that values contributions by non-monetary measures.

Open Source Development can start as an individual practice of few developers, at no significant cost. However, increasing development capacity requires integration effort. You may only start in-house, especially if you make money from keeping your code secret. A counter indication would be an organization that has no product related in house development, but spends most working time at the customers' site. The key side effect is a cultural change of the organization that will require some slack time in the project schedules. The overdose effects of this cultural change are not known yet.

## Reward the Adopter

To get reuse projects accepted among the potential clients, these must gain an advantage even in early project phases. The financial scenario in the long run includes cost reduction and faster development due to code reuse – on the users side. First or early adopters more often experience trouble like instable software, and they have to personally overcome the "not invented here" syndrome. To make reuse attractive, managers and developers who reuse code must be rewarded in a similar manner as those whose code becomes reused.

Each company and manager has different reward mechanisms that are not worth discussing here in detail. Motivation to reuse is raised when it is socially and financially considered just as worthy and respectfully as being the initial creator. As a lot of developers rate innovation higher than transfer into products, management might compensate by different means.

While the costs to REWARD THE ADOPTER are insignificant at first, it requires involvement of developers and technical management. Balanced rewarding schemes show little side effects like loss of innovative drive, but rather allow different personalities to do their best. A technical side effect is that reuse always couples creator and reuser, which requires a policy for managing multiple combined projects. An overdose can lead to reward-only development mode, or cause the projects to lose focus with respect to their initial goals and the companies' business.

> *A project comprising embedded controllers as well as networked servers was facing serious integration problems. One of the initiated measures to overcome them was to code the shared application knowledge in code that was also shared on all platforms. The different cultures of the subprojects needed to overcome their misunderstandings in order to develop this shared code, which was exactly what made integration possible. While the*

*technical inventors of the shared code were recognized for their quality work, the early acceptors and adopters amongst the developers were recognized for advancing the project.*

## Informal Subduction

When a reuse effort has problems to be accepted by future customers, it needs to explicitly manage the customer relations just as any other business would do. The customer needs to receive the impression to be treated first class. This might be expressed by allowing him to control large parts of the reuse effort. Ignore the formal organization chart for a while, and place yourself informally underneath your customer.

The subduction has to be temporary, and must change after the first few successful projects. By that time, the reuse oriented organization has earned significant merits and a well motivated standing on its own. Also make sure that the subduction is limited to user needs and prioritization of tasks and features. Do not allow any customer to explicitly of implicitly violate the principles of dependencies (only towards the reused software) and stability (reused software needs to be more stable). You may emphasize this by asking with each requested change whether it would possibly help your next customers or next project, and how that might in turn influence the current customers' project.

INFORMAL SUBDUCTION requires buy-in by management and constant effort from their side. There are no counter indication, thought it might be wise to keep individual customers' influence confidential, even when it is an in house customer. A side effect is that communication between various levels will improve significantly. An overdose effect occurs when you grant several customers privileges at the same time, which would possibly lead to a steering committee that neither creates a trust relation nor improves communication.

*In a medical devices' company, a large framework project was initiated. To gather expectations and needs, the existing business units were consulted for their requirements and priorities. After the first software could be presented, the one of the product teams with the tightest schedule became most eager to drive the reuse effort. It allowed several developers to join in the framework team, and received more consultancy and insights than the next products in line.*

## Subsequent Customers

You apply INFORMAL SUBDUCTION, but cannot possibly manage to take care for all your customers at once. Especially you cannot subdue under all of them, as the sum of them will steer you in different directions and disrupt the consistency of the software to be reused. Select one customer to be the first one, and work in close cooperation with that one. The other customers are treated as future, subsequent customers.

The selection is a political issue. Talk to all possible customers and ask them: if that project would employ the reusable software, would you also? Try to find the most respected project, and make them co-responsible for the success of the framework by applying to their proud and offering incredible support.

To sequence your customers requires management action that an architect can only suggest. Its costs cannot be exactly given, they are a balance between the side effect of potentially loosing customers that do net feel preferred, and the increased probability to find at least one satisfied and thus funding customer.

## Component Based Development

Large scale reuse efforts aim to compose the future companies' products on a mixture between common code from the organization, and specific code that makes for the individual product. This requires a technical architecture that supports both the ability to integrate a number of software components, and support for independent development of these components.

The standardized interfaces of a commercial available component infrastructure and the technical services provided are important issues that COMPONENT BASED DEVELOPMENT offers. More important than the particular technical infrastructure you choose is the understanding of all participants about the boundaries of the components, and the degrees of freedom they have. Make sure to define how the application structure relates to the technical structure.

COMPONENT BASED DEVELOPMENT requires some educational costs, but it can be start at a single project's scope. Do not apply it when no product is readily available for your environment – unless you understand what it means to build your own component infrastructure [*Völter01*]. A side effect is that you have a chance to reconsider your accustomed and probably insufficient software architecture. When you finally think that everything is a component, you have reached the overdose region and left sensible engineering.

> *The medical division of a large company strived for reuse of common functionality. They decided to develop a component based framework for reuse in various image giving products. Over the years this framework became mature and is now available also to external customers [*Syngo*]. Within the medical business units, the framework is employed in most products targeting the upper market segments.*

## Defer Reuse Profits

When starting a reuse initiative, do not expect it to become profitable in a short to medium time frame. Crafting truly reusable software components, libraries and frameworks is a technical challenge that requires time and experience. Even more challenging is the move to a reuse-oriented organization.

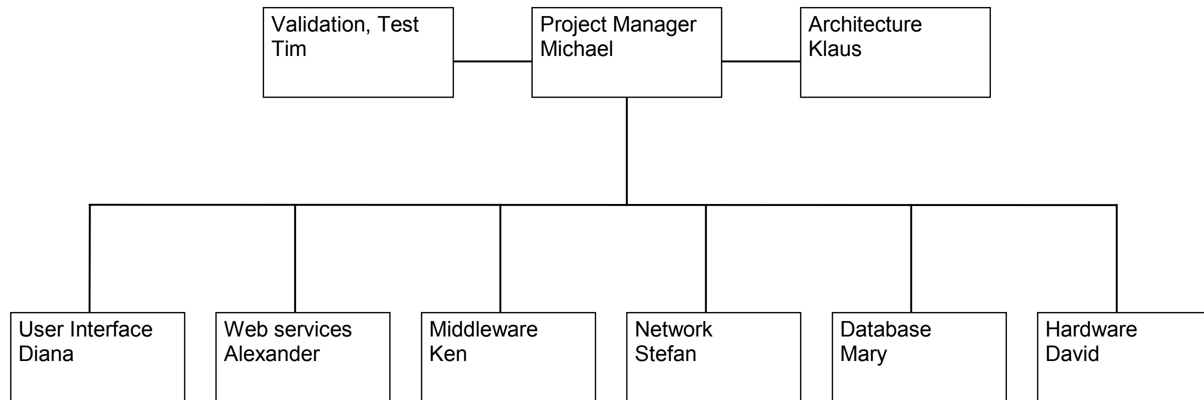
Due to the time required, it would be far too risky to bet your companies survival on reuse. Stick to your overall business goals during the reuse effort. As an architect, do not suggest anything else and rather ensure that the required time is understood and accepted. Allow the organization and yourself some time for this move, and get accustomed to intermediate states.

You should create a shared vision of a reuse-oriented company, but the way and the end result may significantly differ from that vision. Do not stand in your way yourself with overly anxious expectations.

DEFER REUSE PROFITS is a top level management decision that you can only bring to attention, and that might be too expensive for your company. Side effects include jealousy among different groups within the company, especially from those that generate profit for the current business. An overdose of DEFER REUSE PROFITS can lead to the company's extinction.

## Diagnosis: Forgotten Application

*The team is busy with infrastructure and technical details, and neglects to build the application.*




A project is organized along the lines of its technical structure. The different layers and technical subsystems are well defined and assigned to competent developers or teams. The developers bring sound technical knowledge and experience, but have little domain expertise. Except for the project manager who also maintains customer and end user relations, no engineer is familiar with the domain. The project has shown significant speed and successfully reached its milestones, as long as the milestones have been defined along technical achievements. Little or no use case analysis exists, and no user acceptance test is defined.

While each developer is satisfied with his own as well as the team achievements, none of them has a feeling about the state of completeness, or even dares to estimate when the project will be ready to ship. While the vague feeling of incompleteness persists, new team members become quickly absorbed by difficult technical questions. Managers and engineers develop the mutual impression of little understanding, knowledge, or even distrust.

**Symptoms checklist**

☑ The project is organized according to technical issues
☑ The project milestones are defined along technical achievements
☑ The developers do not care for the domain, or are struggling to get it right
☑ You do not know when which user visible function will be delivered
☑ You sense a distrust between technical staff and management

**Diagnosis**: FORGOTTEN APPLICATION or TECHNOLOGITIS

The team and management forget the very purpose of their project: offering a valuable application to their clients. All of the project structure and culture is focused on secondary things, that might be necessary to meet expectations but add no direct benefit.

Another common name for FORGOTTEN APPLICATION is TECHNOLOGITIS, as the entire project is infected by focusing on the technical infrastructure. It might come in specific incarnations such as MIDDLEWAREITIS.

The pathogen is ignorance against the project's purpose, and self-complacency with a selected or assigned task. The project is focused on secondary things that add no direct benefit. Such ignorance can occur when team and management does what it knows how to do best, and this does not match what the user needs most. Over time, everyone is so busy with his own business, that other views and problems are not accepted or taken seriously. Such thoughts dwell merely unconsciously, creating a slight impression of doubt – but in quite some teams it is taboo to address this aloud. The germ often finds a culture medium in human fear.

---

No single therapy can heal ignorance and self-complacency. The suggested medications head in two different directions. Some bring the application and its users closer to focus, others aim to remit the pathogen and appeal to the auto immune system of human nature.

DIRECT CUSTOMER RELATION is in the long-term a curative therapy, which can also be applied preventive. Of the therapies below, it is the least expensive and can be applied on an individual basis without extensive preparation. USE CASE ANALYSIS and HANDBOOK FIRST DESIGN ensure that the development process does not neglect the applications purpose. Given the amount of ignorance though, it may be hard to convince the team to stick to the process or to take its prescriptions serious.

A process-independent improvement comes with the change in organization, as in CODE OWNERSHIP, FEATURE ASSIGNMENT and REVERT THE ORGANIZATION. They are palliative, but nevertheless show a strong effect when combined with a process-related therapy.

A cure is only possible through re-introduction of human ethics like honesty, and an open and fearless communication. As this must come from within the team and the management, the agents can not be applied in a controllable manner. However, you can strive to get the right people into the project by SELF-SELECTING TEAM [*Coplien95*], with some limitations in companies that are really political [*DeMarco+92*]. Improving the communication itself can also help, like in SMALL PROJECT HEAD COUNT, EVEN COMMUNICATION STRUCTURE, ARCHITECTURAL LEADERSHIP [*Coplien+*].

**Therapy overview**

| | **Applicability** | **Effect** | **Related therapies** |
|---|---|---|---|
| DIRECT CUSTOMER RELATION | Any time during the project | Palliative, preventive, possibly curative | Also helps to improve internal communication |
| USE CASE ANALYSIS | Early in the project | Palliative | |
| HANDBOOK FIRST DESIGN | Early in the project;<br>on your next project | Palliative, preventive | |
| CODE OWNERSHIP, FEATURE ASSIGNMENT | Any time during the project | Palliative | |
| REVERT THE ORGANIZATION | Very few times during the project | Palliative, possibly curative | Also helps to improve communication. |

**Caveat** All these therapies are beyond an architect's scope. You can only suggest them or, for all but the organizational therapies, apply them for your own tasks.

## Direct Customer Relation

All project stakeholders are given the opportunity to discuss issues directly with customers or end users. Such a direct contact helps to avoid misunderstandings and can create a climate of mutual acceptance.

It can be hard to take the first steps towards DIRECT CUSTOMER RELATION in an ongoing project. On an individual basis, a simple recipe is “just call and ask”. If you need to bring entire teams together, you might consider social events – but be prepared that these may even deepen an existing gap if no common ground can be covered.

Do not expect a customer in personam sitting on-site with your project. Although this might be feasible [*Beck99*], that person will be set in the middle of conflicting forces. After several months, such an on site customer might be an outsider both to the project and to his own company [*xpforum*].

Some companies do not like to let their customers know about internal issues, and do not trust their own employees to interact with customers in a way the company benefits from this. Whenever strong trust issues arise, you first need to evaluate your own position. When you are in a troubled project, none of the scenarios is really pleasant.

DIRECT CUSTOMER RELATION can start at minimal initial cost and scope. There are no counter indications, but positive side effects on the communication within your project. An overdose effect is that you might run into a variety of inconsistent communication channels that you have to manage.

> *A team of 15 developers of a consulting company implemented an information system that the customer planned to hand to its own clients. For each major application area, and for key technical areas such as the user interface, both parties defined responsible persons that worked in close contact. About half of the developers had frequent meetings with their customer experts, and the entire team came together on special occasions.*

## Use Case Analysis

Establish an explicit and early step in your development process, where you gather the application requirements from a users view. Use cases are the most popular form of such requirements [*Cockburn01*], and can be surprisingly hard to get right. For your purpose you may also use a less formal approach – but do not let go the users' viewpoint.

USE CASE ANALYSIS requires management commitment, and can not reasonably be introduced late in the project. An overdose effect occurs when you spend too much time for the use cases and provide too much detail.

## Handbook First Design

Start the project by writing the user's manual. A serious reviewable draft is sufficient here. This is an elegant way to re-use a specification document in the real world, and furthermore forces you to use a language that each end user of the system should be able to understand. And here is the review process: explain the product concept to end users by showing them the manual, and ask them whether they understand it, and whether they would buy it. (A similar approach is described by [*Pelrine00*])

Though HANDBOOK FIRST DESIGN's costs are likely lower than those of other approaches, it requires management commitment and can not reasonably be introduced late in the project. Very innovative or complex products might be a counter indication, as you need learn and change too much over time – but then, it still has positive side effects on communication within the team.

## Code Ownership, Feature Assignment

Your project needs to cover both technical aspects and domain related functions. You cover the need for system consistency by (relaxed) code ownership on a package level, and focus on domain related progress by assigning features to developers or small teams [*Coplien+*]. Also known as "Function Owners/Component Owners" [*Cockburn98*].

CODE OWNERSHIP, FEATURE ASSIGNMENT comes at no significant cost, but requires buy-in from the entire development team.

> *I admit that I have not seen this work so far, except for very relaxed code ownership. However, it is more plausible than a Matrix Organization – every organization where employees were measured by distinct or even opposite criteria, and different managers, seems to fail in the mid-term.*

## Revert the Organization

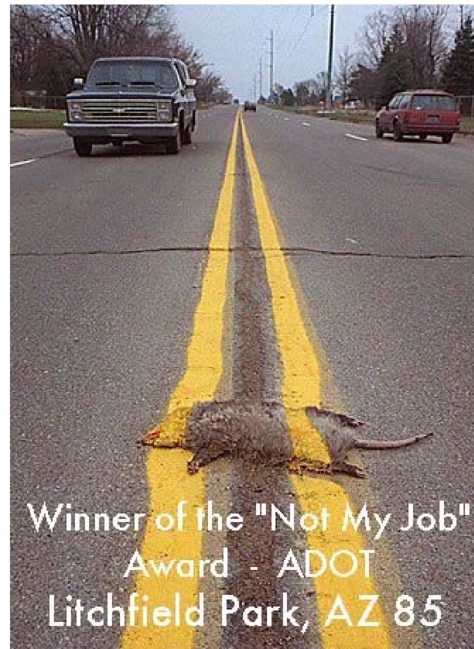
Instead of dealing with all tradeoffs that a matrix organization brings, change your project organization about two times in the projects course. This forces all project participants to think in both dimensions, and gets them used to do so. Depending on the project situation, you still can make good progress in the area of the highest risk. This can easily trade off the restructuring costs.

To REVERT THE ORGANIZATION is a management decision. Its costs are related to its side effects, which include communication changes, irritation, and tighter integration. Do not plan to do it more than twice during the project, otherwise you will cause confusion and introduce management by indifference.

> *The team structure shown in the figure above was chosen intentionally, when a big company started an effort to transfer the existing systems to a new technical platform. The application domain knowledge was present among the team. After the technical foundation was established, the team organization became reverted along the lines of domain functionality.*

## Diagnosis: Ugly Integration

*Developers lead a careless life taking responsibility only for their own code. System responsibility is with persons considered "poor guys".*




You find a well-structured project. Every participant is aware of his/her responsibilities and does a good job in the assigned and accepted duties. Development is oriented along a defined schedule, each task has defined input and output deliverables.

The team has developed a no-intrusion policy, it is banned to look or even comment at somebody else's work. The notion of "ready" is strictly limited to individual pieces of work. Decisions of architectural scope that influence other people's code are hardly accepted and often require management decisions.

Few people care what happens to finished work products: the project lead, and typically a helper to him called integrator, or architect, or assistant. This role is avoided by most developers; a once assigned integrator remains integrator for several projects, and he is working overtime a lot.

New software versions are created scarcely, and a new release or baseline is created in a (at least) two-phase process. First, the developers deliver to the integrator who tries to run the system, resolves conflicts and reports errors back. Second, upon integrator's request the developers deliver their improved version for final integration.

**Symptoms checklist**

- ☑ Clear responsibilities defined for sub-components
- ☑ Strict code ownership, even a "non-intrusion" policy
- ☑ Integrator role is explicitly assigned
- ☑ Integrator alone is perceived responsible for system integration

**Diagnosis**: UGLY INTEGRATION

The project participants avoid taking care for the system as a whole. Integration seems ugly, and does not pay off. The individual developers forget that their project's purpose is a working system, not a collection of timely deliverables.

---

The pathogen is ignorance against the project's purpose, and self-complacency with a selected or assigned task. The individual developers forget that their project's purpose is a working system, not a collection of timely deliverables. The project culture allows most participants to focus on secondary things, and loads the system responsibility on too few shoulders.

Such ignorance can occur when developers work on similar subjects for years and became specialized in them. Over time, in their minds their system part becomes synonymous with the entire system. They resist most attempts to change, because they know that their knowledge is needed within the company. The germ often finds a culture medium in human fear.

---

No single therapy can heal ignorance and self-complacency. The suggested medications head in two different directions. Some bring the application and the system itself closer to focus, others aim to remit the pathogen and appeal to the auto immune system of human nature.

CONTINUOUS INTEGRATION does not remove the germ, but soothes the associated project risk. REWARD INTEGRATION can support the integration efforts and remove some of the ugly aspects of system integration. A possibly curative medication is ROTATE INTEGRATOR ROLE, which in the long term can change people's minds by changing their habits.

The architecture can lay a focus on system integration. An accepted BIG PICTURE ARCHITECTURE can remove a lot of misunderstanding possibilities and act on integration. In distributed teams or systems a COMMON CODE BASE can integrate among different developer groups.

A cure is only possible through re-introduction of human ethics like honesty, and an open and fearless communication. As this must come from within the team and the management, the agents can not be applied in a controllable manner. However, you can strive to get the right people into the project by SELF-SELECTING TEAM [*Coplien95*], with some limitations in companies that are really political [*DeMarco+92*]. Improving the communication itself can also help, like in SMALL PROJECT HEAD COUNT, EVEN COMMUNICATION STRUCTURE, ARCHITECTURAL LEADERSHIP [*Coplien+*].

**Therapy overview**

| | **Applicability** | **Effect** | **Related therapies** |
|---|---|---|---|
| CONTINUOUS INTEGRATION | Any time during the project | Remission possible | Supported by REWARD INTEGRATION and ROTATE INTEGRATOR ROLE |
| REWARD INTEGRATION | Any time during the project | Palliative | Alternative to ROTATE INTEGRATOR ROLE |
| ROTATE INTEGRATOR ROLE | Any time during the project. | Remission possible | More effective than REWARD INTEGRATION |
| BIG PICTURE ARCHITECTURE | Any time during the project, preferably early | Palliative | |
| COMMON CODE BASE | Any time during the project, preferably early | Palliative | Must be accompanied by CONTINUOUS INTEGRATION |

## Continuous Integration

Integrate the system often and early. Make the integration of the entire system a part of the daily or weekly team routine. Establish a delivery process that causes most pain and work at the side of those developers that deliver seldom and integrate least.

The costs related to CONTINUOUS INTEGRATION pay off in the projects lifecycle, and often rather quickly because you detect and correct errors and misconceptions early. It requires involvement of technical management and key developers. There are no significant side effects or counter indications, but you need to adapt the integration frequency to your environment's abilities.

> *A project defined two sub-teams of about 10 developers each. These teams showed significant cultural differences based on different experiences. After the project faced serious integration problems while each team met its own milestones, the integration policy was changed. The entire software was integrated on the target once a week – which is a high frequency for embedded systems. Finally the two teams managed to come to a common culture and work towards a common goal.*

## Reward Integration

Each project has participants that take more responsibilities than they are assigned to. Take care that this enthusiasm does not get absorbed by a "not your business" peer pressure. Reward those efforts that lead to better integration of software components, or better communication structures about integration problems.

While the costs of REWARD INTEGRATION are insignificant, it requires involvement of developers and technical management. An undesired side effect might be that you still find the same people involved with integration as before. A tendency to lose focus due to an overdose has occasionally been observed.

## Rotate Integrator Role

Integration becomes a team task when every team member is responsible for it. As a shared responsibility is hard to implement and may turn into no responsibility, assign this duty explicitly – but to different developers over time. Make very clear that each team member will be in the integrator role on a regular base, and do not rely on accepted responsibilities in this particular case.

The costs of ROTATE INTEGRATOR ROLE are caused by the learning curve of the developers, that they need in order to do the integration. This learning and the associated communication is a desired effect. No overdose effects can occur, as this therapy is a digital one – you either apply it, or you don't.

> *An organization adopted object-oriented development techniques, and all engineers new to it were eager to learn – including the integrator who desired to be released from part of his duties. As some of the experienced developers advocated a certain amount of collective code ownership, the team consequently defined a collective integration ownership. Each week, two different developers of the team became "integrators of the week" and were responsible for the baselines. This worked well because the project team also adopted the policy that code delivered to integration had at least to compile there, which was the*

*responsibility of the delivering developer. The engineers developed a habit of delivering relatively small chunks of work, and did so in a friendly way partly enforced by tools, and partly due to peer pressure.*

## Big Picture Architecture

For successful integration, you need to outline the system and its goals so that the integration can be planned accordingly, and your architecture avoids unrelated development. Define a compact architecture outline that covers the top level of the technical structure and the order and stability of development.

Illustrate the most important issues in a simplified way. The simplifications should match with the developers' experience and scale up to a large extent. Examples are a document-view architecture or a layered architecture. The dependencies among top-level components them must be explicitly modeled, so that the integration can be planned accordingly.

A BIG PICTURE ARCHITECTURE requires time to develop and communicate. There are no counter indications or side effects. An overdose effect can be "micro-architecture", similar to micro-management.

*In lack of a commonly available vocabulary, an architect introduced an extensible architecture with the notion of "colored boxes". Each box represented an extension component, the color indicated its particular purpose with respect to techniques and application. Within each component, a Model-View-Control pattern (MVC) [*Buschmann+96*] came into place, and within the MVC participants one more level of substructure was defined. After some time, the vocabulary and dependencies became obvious to the team, and each developer was able to place a given class at the correct logical location – or to tell what was wrong about it.*

## Common Code Base

To prevent different teams or individual developers from drifting apart, make sure that they communicate and integrate about the same code. Commonly used code bases at the lower end can be class libraries, both bought and build ones. Even more effective are common application classes that integrate at a level beyond technical details.

Such a COMMON CODE BASE brings reuse drawbacks, though. You need to define it early in the project, or initiate a cross-team effort to establish it. The later in the project, the higher the resistance against changes will be.

The costs of a COMMON CODE BASE can be significant, and you need to involve all developers, testers, architects, and managers. Among the side effects is a closer coupling between different parts of the project. The need to uncouple them, like in distributed international projects, is a counter indication. The overdose effects are the same as with all reuse efforts, up to loss of the project.

*The team of the example for CONTINUOUS INTEGRATION decided to limit their integration problems also by means of code. They started to use common type and base classes for both sub-projects, that described the technical infrastructure as well as the key application classes. Thus, it helped to avoid most misunderstanding during integration. The common code base was initiated by an experienced architect, and implemented by a team of four developers from both teams.*

## Conclusion

This article has presented a few diseases of software organizations in depth. They now have a name, are available to examination in a review or assessment, and a number of measures for treatment can be suggested. The reasoning about the causes behind the symptoms allows you to think about further measures yourself, or to tailor the suggested therapies. Most of the therapies can be successfully applied beyond the diagnoses mentioned here, and can possibly be used in a prophylactic way to avoid serious diseases in the first place.

Other works in pattern literature have tried to cover similar needs, though the approach presented here allows for a better linkage between the different therapy patterns and a broader range of different aspects via the more general diagnoses. Jim Coplien and Neil Harrison have written about architecture and organization [*Coplien95*, *Coplien+*], Alistair Cockburn has published project management patterns using a pharmaceutical analogy [*Cockburn98*].

However, the selected diagnoses leave more gaps than they fill so far. Besides the vast amount of obviously missing diagnoses and therapies, they need to cover more field experience and a comparison between alternative agents. Future work needs to collect therapies in distinct areas in order to reach a fair amount of completeness and to explore alternatives in relation to each other.

## Acknowledgements

Many thanks to my shepherd Neil Harrison for his ability to make me think hard, and his willingness to provide yet another credit problem. Further thanks go to the workshop participants at EuroPLoP 2002 for their detailed and thoughtful comments.

## References


| | |
|---|---|
| *Beck99* | *Kent Beck: Extreme Programming Explained, Addison-Wesley 1999* |
| *Buschmann+96* | *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture, Wiley 1996* |
| Cockburn98 | *Alistair Cockburn: Surviving Object-Oriented Projects. Addison-Wesley 1998* |
| Cockburn01 | *Alistair Cockburn: Writing Effective Use-Cases. Addison-Wesley 2001* |
| Cockburn02 | *Alistair Cockburn: Agile Software Development. Addison-Wesley 2002* |
| Coplien95 | *James Coplien: A Generative Development-Process Pattern Language. In: Pattern Languages of Program Design, Addison-Wesley 1995* |
| Coplien+ | *online: http://i44pc48.info.uni-karlsruhe.de/cgi-bin/OrgPatterns.book to be published* |
| DeMarco+92 | *Tom DeMarco, Timothy Lister: Peopleware. Dorset-House 1992* |
| Gabriel00 | *Richard Gabriel: Mob Software. Keynote at OOPSLA 2000* |
| Kerth95 | *Norman Kerth: Caterpillar's Fate. In: Pattern Languages of Program Design, Addison-Wesley 1995* |
| Marquardt01 | *Klaus Marquardt: Dependency Structures. Architectural Diagnoses and Therapies. In: Proceedings of EuroPLoP 2001* |
| Pelrine00 | *Joseph Pelrine: Experience report at OOPSLA 2000* |
| Senge90 | *Peter Senge: The Fifth Discipline. Doubleday 1990* |
| Senge99 | *Peter Senge, Art Kleiner, Charlotte Roberts, Richard Ross, George Roth, Bryan Smith: The Dance of Change. Nicholas Brealey Publishing 1999* |
| Syngo | *A framework for image giving devices, www.syngo.com* |
| Völter01 | *Markus Völter: Server-Side Components. In: Proceedings of EuroPLoP 2001* |
| Xpforum | *communication within the xp-forum e-group, in German, April 2002* |