Performitis

Klaus Marquardt
Käthe-Kollwitz-Weg 14
23558 Lübeck
Germany

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

Copyright © 2003 by Klaus Marquardt.
Permission is granted for the purpose of EuroPLoP 2003

Large software systems and projects develop a high degree of internal complexity by their sheer size, and are bound to trouble. Experienced software architects have learned to detect problems before they have become critical, and apply a variety of different measures in different situations. This knowledge of early symptom recognition and selection of a reaction can be collected in the form of diagnoses and therapies.

PERFORMITIS is a kind of architectural approach that arises from an overly narrow focus on performance during development. While it appears a technical issue at first, closer examination exhibits that its foundations are people and process issues. Accordingly, PERFORMITIS should be treated by technical as well as team and process therapies, where a combination of both is typically most successful.

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. Nowadays, agile development suggests that most of the architecture is defined while the system is already under development.

In many projects, few if any participants can view and understand the system from different perspectives at the same time, and find a common language with developers, managers and other stakeholders. A technical key person in such a position is typically called architect, and fills both technical and political roles. An architect is able to detect technical, structural, and organizational deficiencies, predict their impact on the project, and has an attitude towards solving the underlying problems.

Common views on software development use the metaphor of civil engineering for the architect, which works well for a blue-print like approach. This matches up-front work and the visionary work, but it does not hold for the problem solving and integrating attitude an architect needs to complete a project. Complementing architecture with a different metaphor offers new perspectives for this broader scope: architects work similar to medical doctors.

The software architect is encouraged to take a role similar to a medical doctor. He examines the system for symptoms, makes a diagnosis, identifies the underlying causes, starts with a treatment and continuously adapts to effects and changes. This work approach also matches nicely for projects practicing agile development and for projects in crisis, as most doctors are primarily called in these situations.

To a "medical" architect, a catalogue of diagnoses, examination techniques, medications and therapy schemes can be a great help to spot arising problems and give hints to effective countermeasures. As in medicine, a successful therapy scheme requires constant monitoring, refining the diagnosis and adapting the therapeutic measures to the patient's state.

The patterns in this submission are part of a larger effort to collect existing knowledge in the form of diagnoses and therapies. Some have been workshopped at previous EuroPLoP and VikingPLoP conferences (see the respective references).

The specific pattern forms used are explained in the appendix. All diagnosis names are written in UNDERLINED SMALL CAPITALS and all therapy names in SMALL CAPITALS. Diagnosis and therapy names used but not listed in this paper are marked (↗) and can be found in the references.

Diagnosis: Performitis

Every part of the system is directly influenced by local performance tuning measures. There is either no global performance strategy, or it ignores other qualities of the system such as testability and maintainability.

Tuned Trabant 601 “Rennpappe”, GDR 1986, 19kW [*Trab03*]

The developers use every opportunity to optimize their code, and are eager to discuss different performance measures with their peers. While each single measure may be perfectly justifiable, check whether you observe the majority of the strategies from the following list:

- Usage of C or C++ and its language specific low level features.
- Extensive and early usage of language specific constructs that favor local performance gains over other qualities like encapsulation, reusability or portability. Examples would be public attributes, or inlining (C/C++).
- Avoidance of indirections, on the expense of tighter coupling, limited extensibility and increased effect of individual changes. Examples for indirection range from virtual functions to entire encapsulation layers.
- Keeping control on implementation details by preferring hand written code over advanced language or library features. E.g. extensive usage of pointer arithmetics instead of advanced specific data types (C/C++).

- Avoidance of code libraries that are not controlled within the organization, due to the expectation that its performance will be less optimal than when implemented for the specific project.
- Responsibility of code modules is clustered according to execution threads rather than to consistent logical responsibilities.
- Wide-spread usage of environment specific features, like direct calls to OS, DB triggers and stored procedures, linker directives, or scheduling priorities.
- Application specific code contains knowledge of technical issues and their optimization. Examples would be knowledge of network package size and frequencies, or database structure and joins.

These techniques are applied by more than a small minority of the programmers involved, and all over the source code. Trying to introduce other qualities that are oppressed by the performance tuning measures would cause expensive changes to large portions of the code.

The most significant observation is that there is no overall evaluation or strategy in place that determines where and when which of the above measures are taken. All decisions for tuning measures are made individually on a local scale.

The project is staffed with experienced developers that are familiar with the domain and the task at hand. This is not the first project the key team members work at, and they have learned some important lessons in their previous projects. The key developers all agree that performance is the single property that is hardest to achieve, and that it could cause a project to fail even when it is presumably almost completed. They exhibit a strong desire to get the performance aspects right first, paying merely lip services to other quality aspects.

When new colleagues join the project, or some contractor gets insight into the system, discussions about the way to ensure performance and other intrinsic qualities will arise. These discussions tend to become emotional quickly because they are at the heart of the individual working style and value system – and they could potentially exhibit deficits in large systems design. PERFORMITIS is only resident in projects that are either closed against outside influences, or have developed mechanisms to terminate discussions that raise inconvenient questions. Look out for the closed project society (if you have the chance).

PERFORMITIS infected systems are unstable with respect to technology or requirement changes. Due to the structure ignoring logical separations, the

changes have a large impact and are costly and timely. The effort related to such changes sometimes exceeds the effort required for initial feature development. Look out for fear of changes and explicitly scheduled, often postponed technology updates.

The negative effects of PERFORMITIS are also visible to upper management. For all but the most experienced teams the system becomes instable with respect to delivery dates. Performance oriented development neglects qualities like testability, and creating tests is an expensive endeavor. The project typically provides little tests on code level and integrates late in the development cycle. The spread-out local tuning measures make it difficult to fix the problems that arise with testing and integration while preparing the delivery. Irony has it that the measures taken then may dramatically degrade performance. Look out for more than one seemingly surprising schedule slip immediately before releases, or for an extremely high effort in test and integration of limited functionality.

Symptoms checklist

- ☑ A number of different practices is used to increase local performance on the expense of other qualities
- ☑ These practices are not limited to a dedicated portion of the code, but spread all over the project
- ☑ There is no strategy which local tuning practices are applied where
- ☑ Performance is considered the by far most significant system quality
- ☑ The project closes itself against external influence
- ☑ Changes occupy large parts of the schedule and are frequently postponed
- ☑ Either a lot of effort is spent in tests and integration, or several milestone dates have not been met with very little warning time

Diagnosis: PERFORMITIS[1], also known as PERFORMANCE BLOAT

Not all of these symptoms are unique to PERFORMITIS, and some only become visible in late states of the disease. Early and sufficient signs is the combination of spread local optimizations, lack of a global strategy, and the team attitude.

---

[1] A term closer to the medical nomenclature would be “performance related softwareosis”. Any chronic disease is called an ~osis. Furthermore, it is not the performance that is infected but the system itself. However, “performitis” is a more popular name.

The pathogen is the limited experience of key developers. Humans tend to remember their failures better than their successes, and many developers have learned particular painful lessons. Mostly in distributed or embedded systems, project can miserably fail due to lacking performance – which none of the participants will ever forget. The limitation in their minds is like a Pavlov reflex that happens especially to developers with extensive experience in a particular domain only. They are bound to ignore that systems are solutions that need to balance different qualities.

Caveat There are rare systems where the absolute dominance of performance is not pathological but a conscious and justifiable decision. Hold on a minute – most likely this is not your situation. To find out, make your priorities of different qualities explicit as in the VISIBLE QUALITIES therapy. Even if you are there, these technical therapies offer some suggestions for improvement: PERFORMANCE-CRITICAL COMPONENTS, ARCHITECTURE TUNING, and MEASUREMENT-BASED TUNING.

Note that while PERFORMITIS appears to be a technical diagnosis at first, its real causes are with people and their socialization. While the technical symptoms can be attacked by some therapeutic measures or suppressed by extensive processes, these require continued effort and can at best maintain a state of remission. Curative therapies need to address the pathogen.

The first set of therapeutic measures address the technical symptoms from an architectural level. Separating the PERFORMANCE-CRITICAL COMPONENTS is the basic architectural technique to avoid performance bloat. Selecting the most efficient places for performance tuning is the topic of ARCHITECTURE TUNING.

Other therapies intend to limit and control the effect that performance gets by explicitly assigning room for performance in the development process. VISIBLE QUALITIES allows making a case for performance, but opens room for other qualities as well. A TEST-ORIENTED PROCESS may soothe most of your symptoms, especially when you prepare for a release. TIME BOXED RELEASES allow for healthy little time to early, detailed performance considerations. MEASUREMENT-BASED TUNING can become a relief from thinking too much about tuning up-front and probably at the less relevant places.

Some therapies can only come into place when some of the developers are open-minded and willing to learn. REVEALED SUPERSTITION is an intellectual way to try to change the developers' habits. ARCHITECT ALSO

COACHES complements this for developers who can stand somebody working closely together with them.

Infections in living organisms cause a kind of fever that is meant to attack the pathogen, even before it is identified. In PERFORMITIS infected teams, experience from other cultures causes such a fever. Discussions about the way to ensure performance and other intrinsic qualities will arise. These discussion cause friction and result in a learning process that helps to improve the balance between different qualities. Feverish therapies are STAFF EXCHANGE and DEDICATED ARCHITECT[2]. Be aware that high fever might well kill an organism, and that its degree needs to be controlled.

Combine the feverish therapies with any process or technical therapy of your choice. No combination of the suggested therapies can be harmful to the project, they all have mutually increasing effect. However, too many therapeutic changes at the same time might break morale and the team structure. Take your time to introduce one after another, and anticipate which one brings the most effect in the current situation.

Therapy Overview

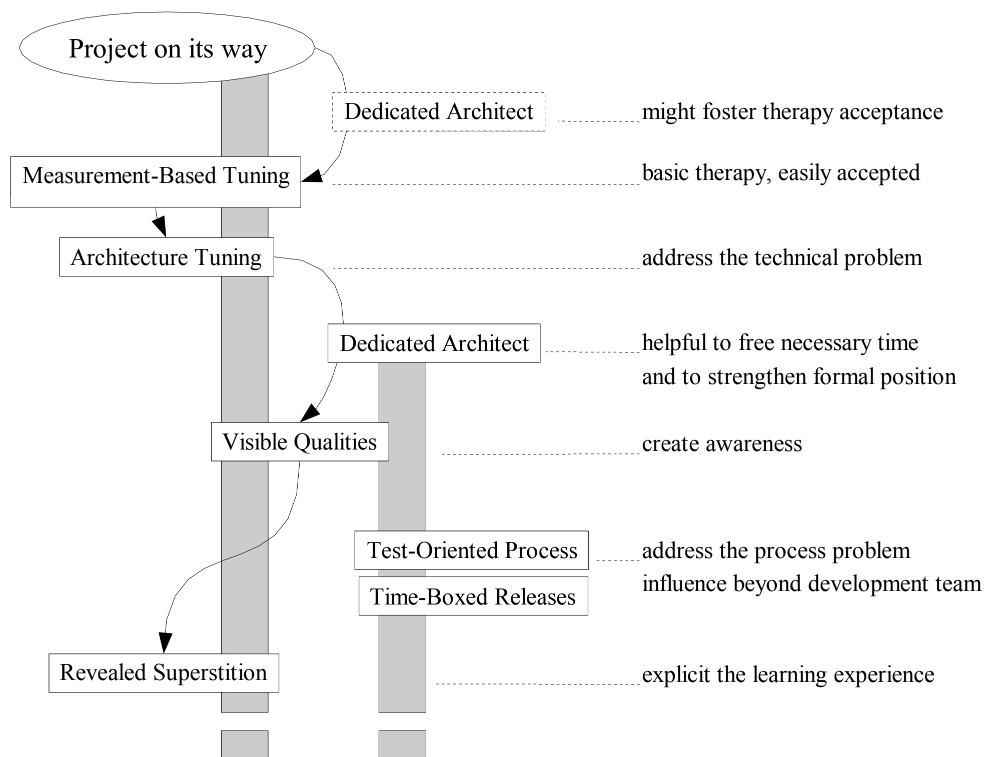
	Applicability	**Effect**	**Related therapies**
PERFORMANCE-CRITICAL COMPONENTS	Early in the project	Preventive; remission possible	Works best with a DEDICATED ARCHITECT
ARCHITECTURE TUNING	Any time during the project	Remission possible	The essential strategic performance tuning
VISIBLE QUALITIES	Preferably early in the project	Preventive; remission possible	Works best with a DEDICATED ARCHITECT
TEST-ORIENTED PROCESS	Preferably early in the project	Palliative; remission possible	
TIME BOXED RELEASES	Preferably early in the project	Palliative; remission possible	Combine with TEST-ORIENTED PROCESS
MEASUREMENT-BASED TUNING	Any time during the project	Palliative. Preventive when applied early	Works best with a DEDICATED ARCHITECT
REVEALED SUPERSTITION	Any time during the project	Remission possible	

[2] While these measures are surgery rather than infection, its effects on the team are more adequately described with the fever metaphor

	Applicability	**Effect**	**Related therapies**
ARCHITECT ALSO COACHES (EMPHASIZE ~ILITIES)	Preferably early in the project. Whenever team composition changes	Remission possible	Requires a DEDICATED ARCHITECT
DEDICATED ARCHITECT	Preferably early in the project	Remission possible	Combine with process or technical therapies
STAFF EXCHANGE	Any time during the project	Curative, or remission possible	Combine with process or technical therapies

Suggested Treatment Schemes

Provided the project is already well under its way, you should focus on therapies that can be used any time during the project.



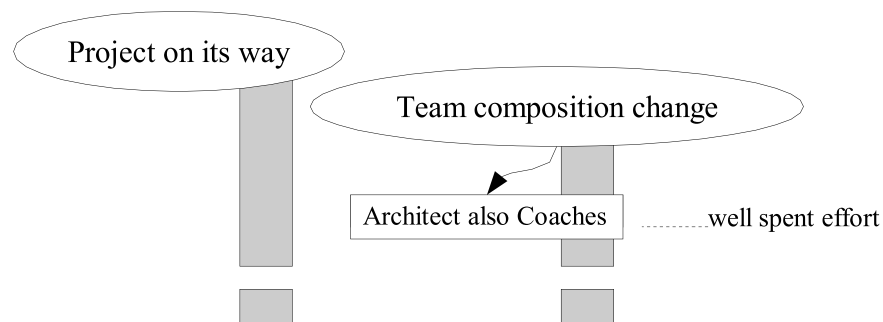
If you already are in a strong inter-personal position, you may even skip DEDICATED ARCHITECT. MEASUREMENT-BASED TUNING is one of the easily accepted. When you start ARCHITECTURE TUNING on basis of the gathered

observations and their interpretation, you can prove to the key developers that you take performance serious and cover their concerns.

The following steps depend strongly on whether you perceive a growing personal acceptance. If you fail to get that acceptance, you should consider addressing a missing DEDICATED ARCHITECT very soon. In case the key developers still ignore you, consult with technical management and suggest process changes to them that help them reach their goals, like TIME BOXED RELEASES, possibly on the expense of the developers' motivation. Alternatively, try to influence QA to demand a TEST-ORIENTED PROCESS. You need to consider that such a therapeutic schema against the team's adopted habits might lead to a STAFF EXCHANGE rather early, and possible drawbacks will be attributed to you.

With growing personal acceptance, you can in parallel introduce VISIBLE QUALITIES to start a team learning process. Over time, those developers eager to learn will notice the REVEALED SUPERSTITION between the different qualities, and will have learned for their professional life. Depending on the learning curve, be relaxed on the timing you care for the VISIBLE QUALITIES. It might be OK to check them with every release only.

Whenever the team composition changes, take your time to coach new members for a while, and EMPHASIZE the ~ILITIES of system architecture.



Performance-Critical Components

Applies to projects in domains that require high system responsiveness.

A development team that is aware of performance tuning needs to prepare the tuning measures before they are actually done.

It is hard to foresee where changed will become necessary later in the project, …but preparation early in development saves restructuring effort and time later.

Performance can be attacked on every level of development, …but initiatives on architectural level likely have the most impact.

Incremental development can proceed fastest when the user-visible functions are developed independently, …but shared libraries and common infrastructure can become more mature and efficient than dispersed items all over the system.

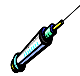
Therefore, factor out the performance critical components, and limit preventive tuning measures to these. Start by dividing the system according to ↗DIVIDE ET IMPERA in a way that business logics, display, distribution and technical infrastructure are independent of each other and can be tuned individually. Ensure that all applications use the same infrastructure so that central tuning measures become possible.

In systems that have been found to waste performance, these areas were good candidates where a responsiveness gain by an order of magnitude was possible:

- Inefficient use of system resources or infrastructure.
- Multiple repetitions of identical calls without functionality gain.
- Badly designed or agreed interfaces requiring multiple data copying or data format conversion.
- Extensive tracing and logging; error handling strategies that pollute the standard flow of operation.

To avoid these pitfalls, your system needs to be prepared to change all internal access strategies and rearrange all call sequences. Make sure that you can start with ARCHITECTURE TUNING whenever you need to. Refactor to the degree that all changes can be made most easily [*Fowl99*]. To avoid large and late refactoring efforts, start with a piece-meal growth approach and a "clean desk" policy that includes refactoring in the daily work.

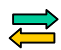
Within each of the components above, again distinguish between published and internal functionality. Factor everything specific to your particular environment into distinct components, like services, calls to technical services, database queries, and handle acquisition. Make sure that the responsibilities among all components are clear and concise. Separate the logical contents from the physical execution, with few explicit linkage points.



PERFORMANCE-CRITICAL COMPONENTS is a preventive therapy that increases the system's adaptivity to further therapies such as ARCHITECTURE TUNING. It fosters many qualities, but does not directly affect performance in and by itself.

The entire development team and technical management need to be involved. PERFORMANCE-CRITICAL COMPONENTS increase the costs you would spend on architectural decomposition. Though the initial costs will pay off if you really run into performance problems, consider PERFORMANCE-CRITICAL COMPONENTS as a risk reduction strategy.

Do not apply PERFORMANCE-CRITICAL COMPONENTS when your system is very small, or applies standard technology only. Though you gain valuable experience, the costs hardly pay off in these cases. Another counter indication is a weak position of the architect in the project. Side effects include an improved logical structure of your system. Overdose effects would be ↗DESIGN BY SPLINTER if you drive the separation to an unbalanced extreme, or a micro-architecture similar to micro-management violating your ↗DEFINED NEGLECTION LEVEL.



PERFORMANCE-CRITICAL COMPONENTS is the preventive strategy to later ARCHITECTURE TUNING. It can be one of your risk reduction measures applied with VISIBLE QUALITIES. To motivate PERFORMANCE-CRITICAL COMPONENTS, it can be wise to REVEAL SUPERSTITION first.

Start with a useful separation that is most likely to support your actual needs. Keep the performance critical components as small as possible by iteratively repeating the division.

From the experience the team has gathered in the domain, both application and technology domain, you already know which parts of the system are likely to become the bottlenecks. Run a retrospective to uncover which areas have been performance relevant in previous projects. You will experience a fair amount of support when you separate these from the remainder of the system, and apply ↗EXPLICIT DEPENDENCY MANAGEMENT.

In a real time patient information system, all transient data was stored in shared memory. This was hidden from the clients to this data; some opaque access classes provided an interface independent of implementation issues. Tuning the performance of data access and locking granularity was located within the access layer classes.

Architecture Tuning

Applies to projects that need performance tuning.

A development team experiences severe performance problems, and needs to decide how to tackle them.

Local tuning efforts based on profiling help to improve the system's responsiveness, …but you'd need to have a lot of local improvements to push the overall performance by orders of magnitude.

Performance can be attacked on every level of development, …but initiatives on architectural level likely have the most impact.

Therefore, tune the system on architectural level. Before you initiate any other improvement efforts, make sure that the architecture itself helps the system responsiveness, and exhibits no obvious flaws or "black holes" in which processing power vanishes.

Ignoring the architecture level might get you lost in an endless sequence of severe battles. Your actual goal is not only to win the performance war but to win your peace with performance.

Looking at the architecture, experience from large projects shows that performance is mostly decreased due to one or more of the following mechanisms:

- Inefficient use of system resources or infrastructure.
- Inappropriate locking of shared resources, or inappropriate transaction granularity.
- Multiple executions of identical calls without functionality gain.
- Extensive network or inter process communication.
- Blocking operations in tasks supposed to be responsive.
- Inappropriate distribution among clients and servers (e.g. query result sets).

- Inappropriate database schemas (too little or too much normalization).
- Extensive error checking, tracing and logging.
- Error handling strategies that pollute the standard flow of operation.
- Interfaces requiring multiple data copying or data format conversion.
- Doing everything doable as soon as possible, or as late as possible.

A few changes to the architecture or top-level design can increase the performance by orders of magnitude. Performance tuning typically follows a few fundamental principles [*Marq02c*].

If your system does not allow to implement the changes you have identified being necessary, you need to refactor it in advance. Typical refactorings for tuning the architecture are the introduction of shared technical components, separation of resource maintenance from resource usage, and separation of performance critical tasks into distinct components.

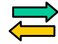
While the system is developed, it is good practice to make these late changes as convenient as possible. As PERFORMANCE-CRITICAL COMPONENTS explains, this is most efficiently done by maintaining a design with a clear separation of responsibilities, and a dependency structure with no (if any) compromises. Such a structure also helps during development with respect to testing and task assignment, and to maintenance in later project phases.

ARCHITECTURE TUNING is a process to find curative technical solutions to technical symptoms. If the related diagnosis indicates that the pathogen is beyond the technical scope, it can lead to remission.

ARCHITECTURE TUNING involves the architect and every developer assisting in analysis and implementation of the performance tuning. Its cost are hardly predictable, they depend on the necessary refactoring effort and the actually implemented changes to the architecture. However, in large projects they are lower than the costs of numerous attempts for local optimizations, and it is more likely to be effective.

There are no counter indications to ARCHITECTURE TUNING – given that you do not consider to abandon tuning or the project at all. The

side effect is a well-structured system. No overdose effect is currently known.



ARCHITECTURE TUNING comes with the least cost when you separate the PERFORMANCE-CRITICAL COMPONENTS early in the project.

"A contractor had managed to become the mind monopole at one of my customers. He motivated his queer data model with reasons such as 'using DB/2, comparing integers shows higher performance than comparing strings'. (The effect exists, but can be neglected compared to the costs of disk access or joins.) When the system went productive, it needed 1.7 seconds per transaction, which would have caused annual operation costs of several 100,000 €. Tuning measures saved around 10-20%, but to reach a performance comparable to similar systems a factor of 100 would have been necessary. I was asked to evaluate the architecture for optimisation possibilities, and found sufficient opportunities to save a factor of 1000 – starting from simple measures like skipping consistency checks of large data structures with every internal call (the much too large structures contained several 100 sets of data), up to fundamental changes to the architecture."

Visible Qualities

> Applies to projects whose team focuses all work and thoughts on a few essential ideas, but ignores all other issues that might also be or become important to the project's success.

In a development team that focuses on a particular quality of the product, you need to address further important system qualities that are essential to adequately manage the system architecture.

Neglecting internal qualities can cause a large system to break under its own weight,	…but the value they add to the software is hidden and becomes visible only in the long term.
Internal qualities can be crafted intentionally into the software,	…but they are hardly visible from a bird's eyes perspective.

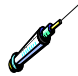
Therefore, make your system's internal qualities visible. Similar to sound risk management practice, maintain a list of your top five qualities. Define measures to achieve them, and determine frequently to what extend you have reached your goal.

The key issue is to raise awareness for the existence of these qualities and their relative importance in the team and in management. Especially when the internal system qualities are unbalanced, ask the team come for a list of possible qualities and discuss their value and advantages. The team should order them according to their priority. Do not mind if your favorites are not the topmost – you will go through the list every week or two and re-evaluate.

Do the same process with the management, and make both lists visible. While it is often not possible to resolve any conflict and come to consensus, the fact that all qualities are there and considered important leads to awareness, a more careful balancing and to an architecture and design that addresses different qualities explicitly.

You need to maintain the lists, find criteria how to evaluate whether a specific quality has been achieved, and define appropriate actions [*Wein92*]. This could become a part of a periodically scheduled team meeting. Especially the evaluation criteria would be a tough job, as most qualities show only indirect effect. Try to define goals that appear reasonable to the

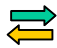
project. If you or the team fails to define criteria, leave that quality at the end of the list for the time being.



VISIBLE QUALITIES is effective through creating attention and a positive attitude. The attention achieved by the top-five list causes second thoughts, awareness, and potentially actions, while the measurable achievement fosters a positive attitude that in itself already could improve the quality of work.

The work and initial costs are with the architect, but VISIBLE QUALITIES requires involvement of the entire team. In the mid term, the effort required is comparable to mentoring or coaching, while in the long term it pays off through improved development practices.

There are no real counter indications to VISIBLE QUALITIES, but if your team is resilient to learning other therapies might be more cost effective for your project at hand. You might experience negative side effects if you fail to explain the importance of different qualities, and a continuous neglection of specific qualities might finally break a large system. Prevent this by establishing a veto right on certain priorities. An overdose could be injected if the team does not get the idea at all, or is disgusted by the somewhat formal process. Use the drive for discussion to come to an adequate dosage.



If you look for less formal approaches, look out for a ↗MENTOR or apply ARCHITECT ALSO COACHES. VISIBLE QUALITIES are successfully accompanied by ↗ARCHITECT ALSO IMPLEMENTS and REVEALED SUPERSTITION.

For motivation of the team and management, the testability quality often is a good starter. Its benefits towards risk reduction and customer satisfaction are obvious, and it can be verified with concrete actions, namely implementing the tests. For testability, the achievement criterion could be “all classes are accompanied by at least one unit test” or, if you introduce unit tests late in the project, “every fixed defect has to be accompanied by at least one new test case”. If for some reason the unit testability is hard to achieve, this is a potential hint for a design fault. To get away with a rule violation, a developer should need to convince the architect. There are situations e.g. in GUI development that are hard to become unit tested, but

improvement suggestions may enable to test at least parts of the functionality, e.g. after a class has been split in distinct parts.

It is not important to maintain the list for a long time. If you introduce it, and hold it up often enough so that the developers know that you are serious about it, you might neglect the list and only check it at the start of a new iteration or release period. The check to what degree you have reached the internal qualities never becomes obsolete, but can be reduced to one check with each iteration or release.

Some qualities are hard to measure by numbers. For the measurement of few qualities commercial tools are available. As an example, the software tomograph [*Lipp04*] supports a quantitative evaluation of the internal software structure.

> *"The team was new to object-oriented design, so we discussed a lot about the promised qualities it should deliver. We started to do ↗JOINT DESIGN at the white board, and I showed on some examples how a high extensibility could be reached, how testability could be increased, and what amount of decoupling required what effort. When the team size increased, ↗DESIGN REVIEWS became an essential part of the project. Initially I participated in most, and we established an ordered catalogue of criteria to check. With this catalogue, the process was accepted and carried by the team. Closer to the end of the project, the team decided to focus on other issues and reduce the formality of the design reviews. By that time, the project lasted for more than two years; all team members had significant expertise and shared a common sense."*

Test-Oriented Process

> Applies to projects whose major implementation decisions are derived from the developers' guts feeling. No effort is made to find out whether these decisions are appropriate.

In a development team that is overly concerned about specific system properties, expensive measures are taken without evidence of necessity or possible prove of effectiveness. Important aspects of the project goals are hidden behind the self adopted blinders.

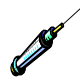
Experienced developers have gone through extensive learning processes,	…but every new project comes with problems you have not yet experienced.
It is unlikely that somebody learns from theoretical lessons without a match in the daily practice,	…but each project has stakeholders whose practice differs from the developers world and promotes further foci.
Changing the development methodology can be much harder than changing some technology,	…but tackling a process issue with process actions causes the least friction.

Therefore, focus the development on testable achievements and create a test for every development step. Consider no functionality completed until it has been tested. Explicitly include things that are hard to test, like the architecture, and the system performance.

Most blinders are based on previous experience. Typically the developers can well describe what the actual problem back then was, how it has been discovered, and what they would make differently to avoid similar problems in future projects. From here, defining some kind of test is usually a small step. Messed dependencies can be tested by evaluating generated dependency graphs; insufficient or unfavourable architectures can be tested by reviews according to self defined criteria; failed performance goals can be tested by load tests and frequent profiling against estimated thresholds. If

a blinder cannot be expressed in terms of some test, it is probablyjust a prejudice instead of an experience.

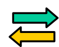
When changing the project's way of working, make sure that the persons with the greatest level of concerns feel as winner. If possible, push them into suggesting tests themselves. Never insist that the idea comes from you, and never fully expose why you were so eager to introduce tests.



TEST-ORIENTED PROCESS is effective against all diagnoses that stem from process related blinders. Its mechanism is to base all experiences on a common currency called "test" – similar to business decisions that require all known influences to be converted into "money" for comparism. This offers a global view on a lot of different aspects of development, including missing ones as well as blinding ones.

Changing the development process requires buy-in from all project stakeholders. Like all changes to the way of working, the key costs are proportional to the resistance they create. Agree on compromises that help to reduce resistance. Suggest the changes early while the team is still small, and the blinders have not had a severe effect.

The only real counter indication against TEST-ORIENTED PROCESS would be to introduce it near the end of the project, and it is not definitely sure that it will fail without drastic measures. The side effects are the reason you introduce it in the first place: attention to previously blind spots, and time and effort spend there. The overdose effect would be to test to an extend that is no longer cost effective – but you are not very likely to experience it.



TEST-ORIENTED PROCESS goes well together with all other therapies that do not change the development process. If you need to introduce other methodology changes as well, like a focus on architecture, it might be appropriate to find the common ground among the changes first, and then change the process according to which risks are the most important ones in the current situation.

This appears to be the obvious thing for every software developer who has seen some kind of development methodology. Testing is the broadest intersection between waterfall-like and agile ways of thinking.

Management and quality control will appreciate your initiative. The project development progress becomes visible and easily traceable, while the releases inherently bring a minimal level of quality that makes formal testing much easier.

> *"We started to introduce unit tests for each completed functionality that was available on multiple platforms. These helped to limit the negative effect of changes to a common code base that were impossible to test on all systems employing the code with reasonable effort. Performance was an issue, so we expanded the test framework with time measurement functionality. A functionality that once caused a performance problem became accompanied by an additional unit test with no functional test criterion, but with an execution time criterion. The code had to satisfy the timing requirements of the most performance limited client, otherwise the test was not considered passed."*

Time Boxed Releases

> Applies to projects that consume their time by preparing for some future event that might never come, thereby violating some practices of sound software engineering.

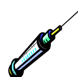
In a development team that has lost focus of the project's purpose or the key architectural solution issues, you need to reinforce the main objectives of the project.

Former experience leads to anticipation of problems to come,	…but when the problems do not come, you loose the effort spent on anticipation.
Preparing for future possibilities requires time,	…but the time is best spent on the problems you have at hand, and on increased customer value.
Time pressure reduces the willingness to spend effort in quality work,	…but adequately balanced internal qualities enable the team to proceed faster, and cope with yet unforeseen situations.

Therefore, schedule the releases of your software in frequent, small intervals. Convince your management that sticking to the release dates is of major importance, just as important as keeping the internal VISIBLE QUALITIES.

When the team remains unchanged, the only variable you allow for negotiation is the scope, the expected functionality contained in each delivery. However, all project stakeholders will strive to have as much usable functionality with each release. This leads to a strongly perceived lack of time to care for tiny details early in the project, including early tuning measures except when well motivated as MEASUREMENT-BASED TUNING.

———

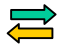


The mechanism of TIME BOXED RELEASES is to focus the development team, and spend only effort in those tasks that pay off

within the next weeks. Implicitly, everything that has only a vague chance to pay off will not be started unless all alternatives are worse.

TIME BOXED RELEASES require management decisions and affects the entire team as well as other project's stakeholders. Its costs are comparable to other substantial process change costs, and are not caused by the mere measures themselves. Due to reduced project risk and less effort spend in vain on irrelevant topics, it probably pays off quickly

Counter indications to TIME BOXED RELEASES are violations to the entry conditions. Internal qualities may be hard to achieve when under time pressure, and the temptation to ignore them is high. Do not start with TIME BOXED RELEASES unless you have established some taboo on the internal qualities. Side effects are the desired emphasis changes of the projects. Some developers might feel uncomfortable with the increased time pressure, with their personal ways of reaction blocked, so prepare for some demotivation and help to establish a fearless environment. Overdose effects are reached when the time boxes are so small that your development environment does not allow for significant achievements, or when you fail to mitigate the side effect risks and induce stress and fear to individual team members.



Accompany with quality oriented process measures. It is necessary to ensure that the VISIBLE QUALITIES are established and taken seriously. A TEST-ORIENTED PROCESS helps with the necessary frequent integration, to measure progress, and to achieve internal qualities. Introduce MEASUREMENT-BASED TUNING as the standard way to motivate tuning measures.

"TIME BOXED RELEASES are a two-edged sword. I have been at a team that was forced to implement features, and ignore performance for a long time. When performance had degraded to a degree that the customer could no longer reasonably execute his own tests, the next iteration became dedicated to performance tuning alone. TIME BOXED RELEASES are a cure for many effects, but you might experience situations where unintended scenarios emerge. In that project, we were ignoring performance issues for too long."

Measurement-Based Tuning

Applies to projects in domains that require a high system responsiveness, when the team is only partially familiar with the domain and its specific requirements.

Every developer knows that tuning the system for performance is necessary. The key decisions are when to take measures, and then to know which tuning measures are most effective.

Measures taken early are typically most effective,	…but measures taken on assumptions instead of proper knowledge are often leading nowhere.
Being afraid is always bad advice,	…but being aware of possible problems is wise.

Therefore, measure where the real bottlenecks are and start tuning measures there. Do not take preventive measures against assumed performance problems. Spend your performance tuning effort where you know it is most effective.

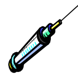
Whenever you assume some problems, turn your assumptions into knowledge. Critical architectural issues can be clarified by spike solution projects [*Beck99*] or prototypes [*Cock98*] with the sole purpose of identifying the actual performance issues. These spikes are most useful when you have established load profile scenarios or performance budgets [*Marq01b*].

Where you cannot gain knowledge for some reason, follow sound practices and "proactively wait"[3]. Resist the temptation to begin with micro tuning. Instead, focus on other qualities, especially on testability and maintainability. Most performance tuning measures on architecture or design level require a clear distribution of responsibilities anyway, and you can spend the structure clean-up effort now, when it hurts least.

To "proactively wait" is an important, but difficult virtue. Key to this technique is not to miss the time when decisive action is necessary. This technique requires self-consciousness and constant observation. When some indication of performance problems becomes sensible, start over and turn

[3] To apply the right amount of waiting is an important virtue of a medical doctor [*Marq99*]. While the symptoms have not become severe and the patient does not suffer, at lot of diseases are left to mere observation until a serious change occurs.

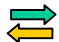
your impression into knowledge. The thresholds when to take action are typically subject to personal taste and working style and require significant experience. However, discussing them openly with colleagues helps not to miss important indications, and the rarely absent lack of time prevents from being overly responsive.



MEASUREMENT-BASED TUNING is not directly effective in a curative way, but frees attention and effort to be put on relevant topics of the project.

All development team members and technical management needs to be involved with MEASUREMENT-BASED TUNING. Interestingly, the costs of MEASUREMENT-BASED TUNING are often negative. It prevents effort being put in mislead measures, and enables you to proceed faster during initial development as well as during performance tuning phases. The costs spend on convincing other stakeholders of the validity of this approach, and of constant observation are typically low.

There are no counter indications to MEASUREMENT-BASED TUNING. The side effects are desired: the team pays more attention to other qualities, measures are taken in an informed manner, and the project proceeds faster. Overdose effects would be to ignore the obvious, common sense in your domain, or to wait too long before you take corrective action.



MEASUREMENT-BASED TUNING can be combined conveniently with ARCHITECT ALSO COACHES, as both therapies emphazise to care for things beyond performance. One of the key practices to prepare for late tuning measures is to separate PERFORMANCE-CRITICAL COMPONENTS.

"When we first used an object oriented database, we put all data in it that needed to be shared between different clients. This design lead to a highly consistent system. Unfortunately, it was also horribly slow. We lived with that fact for some time, hoping that increasing our knowledge about the OODBMS would provide us with counter measures. After two iterations, the GUI team decided to stub the database and leave the process of continuous

integration. This was a severe warning, and we checked for the database performance.

"It turned out that the most expensive data the OODBMS was occupied with was transient data; it was distributed among different clients but did not require persistency. Two concrete actions were initiated. First, the distribution mechanism became separated from the database access. Second, for the sake of consistent class interfaces, the classes meant to become persistent were no longer derived from the OODBMS base class. Instead we provided a distinct persistence service that we passed the objects to, and maintained the database schema by generating the persistent classes from the application's class model."

Revealed Superstition

Applies to projects in domains that require a high system responsiveness.

In a development team that is blinded by a particular experience and tends to ignore or even deny all issues that do not fall into the selected category, you need to address architectural needs that are critical to the success of the project.

People tend to concentrate on a single issue, neglecting everything else,

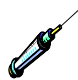
…but a broad overview uncovers connections among different system qualities.

Some external qualities are more relevant to reach than other external or internal ones,

…but in most cases achieving some quality does not necessarily prevent other qualities.

Therefore, identify the relation between different qualities, and separate actual contradiction from developers' superstition. Outline which means would foster which internal and external qualities, and which would prevent you from achieving them. Initiate those actions that help multiple quality aspects of your system and that complement each other. Teach your peers about the assumed or perceived contradictions that are not existent in reality.

In particular, performance and the qualities fostered by separation of concerns can go nicely together. However, you need to separate along the lines that help increase responsiveness – which is probably not the way you would initially decompose a system.

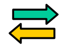


REVEALED SUPERSTITION works by focusing attention at the unspoken assumptions that are blinders to developers, and proving them wrong.

All developers need to be involved, and it is helpful to include technical management as well to avoid contradicting your efforts by restrictive management policies. The costs come from the two

phases of the therapy. Identifying measures and effects to qualities require some preparation that depend on the architect. The ongoing teaching is a mentoring effort that needs to last for some time; its effort is similar to other mentoring or coaching techniques.

There are no counter indications to REVEALED SUPERSTITION. Possible side effects or overdose effects cannot be attributed directly to REVEALED SUPERSTITION.



Combine REVEALED SUPERSTITION with VISIBLE QUALITIES. PERFORMANCE-CRITICAL COMPONENTS gives some concrete examples of frequently perceived contradictions.

The therapy patterns PERFORMANCE-CRITICAL COMPONENTS and ARCHITECTURE TUNING exhibit one of the most popular virtual contradictions. If you choose a clear, concise structure, you are (a) more likely to quickly locate performance relevant issues, and (b) be able to fix them with much less effort. In the end, you get a system that runs faster and shows a better internal structure, increasing testability and maintainability.[4]

> *"Some years ago I worked for a business unit that was supposed to build both a domain specific framework, and the first product based on that framework. My role was product family architect, so I was in close contact to the management and to future users of the framework. Sensing tough decisions, I asked the management for their priorities. Of the choices I offered to them, they selected two things being both on top of the list: quality, and time to market. At first, I was frustrated from not getting a clear priority. After some time I learned that this priority combination strengthened the position of the architecture a lot, and was a perfect motivation to emphasize a healthy, consistent, respected system architecture – the best thing one could do to reach both goals.*
>
> *"Due to political difficulties, the business unit and the idea of a domain specific framework were abandoned later. All projects*

[4] I'd like to recommend some course that teaches about the virtual contradictions, and how different qualities can be achieved synchronously. If you are aware of such a learning offer, please let me know.

were stopped, and the developers had to find new positions in other parts of the company. The intellectual property and most of the framework team became absorbed by the project that originally was supposed to build the second product based on that framework. Released of political pressure, the architecture and the team building started to pay off. The project became more successful than expected. Currently, its results are evaluated for reuse in other products as well, maybe forming the basics of a framework."

Architect Also Coaches

Also known as: Emphasize ~Ilities

> Applies to project teams that focus their work and thoughts to a few essential ideas, but ignore all other issues that might also be or become important to the project's success.

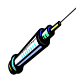
In a development team that has an informal design and architecture process without a dedicated role assignment, and that focuses on a particular issue of development, you need to address further important system qualities that are essential to adequately manage the system architecture.

People's experience is valuable to the project,	…but the experience needs to fit the current project's size and criticality; building a large system requires attention to issues that hardly matter in smaller systems.
Education opportunities for developers are readily available in courses and classes,	…but learning as you do your job is more efficient, and has the potential to teach lessons that you never forget.
An architect's experience and view on the software world is limited,	…but the architect has the broadest view of the developers.
Neglecting internal qualities can cause a large system to break under its own weight,	…but the system can not become any better than the architect and the team know how to build it.

Therefore, the architect becomes responsible to coach the development team about the internal qualities (the "~ilities") that are essential to crafting large software systems. This is an efficient way to succeed in his core duties – maintain the ↗BIG PICTURE ARCHITECTURE, support management and development, balance the different forces on the architecture, ensure consistency, and broaden the architectural view – because it involves all developers and avoids resistance.

This sounds simple, but its implementation requires serious effort. While most project situations can live with a developer informally taking that role, ARCHITECT ALSO COACHES requires significant time and effort. You need to have the architect's role defined and assigned, as in DEDICATED ARCHITECT.

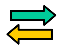
The internal qualities the architect needs to emphasize correspond to the size and complexity of the solution under construction. Testability is a favorite one that pays off quickly. Ensuring testability is closely related to a design with a clear distribution of responsibilities and strict adherence to dependency rules. These two are also needed to allow parallel development of several tasks and potentially several teams. Maintainability is another key quality for large project, as during initial development the first of its components are already being maintained.



The mechanism behind ARCHITECT ALSO COACHES is respect and trust, and spread knowledge. The team will respect an architect that knows the system, has a stake in it, and solves the day-to-day problems. Trust is necessary to learn and to change the own behavior. Spreading the knowledge helps convincing developers and avoiding resistance.

ARCHITECT ALSO COACHES involves the architect and potentially all team members. The costs can become significant because you need to dedicate a lot of time to it. However, the costs for education and consistency will reduce the project risk and likely pay off over the project's course.

If individual developers send signals that they would not accept coaching, this might be a counter indication. ARCHITECT ALSO COACHES has side effects on the work load that the team can manage. It will decrease in the short term, but eventually increase in the mid to long term.



The acceptance of the architect coaching is likely fostered when you apply ↗ARCHITECT ALSO IMPLEMENTS. Pair programming [*Beck99*] or ↗JOINT DESIGN are good opportunities to start coaching. When some team members do not accept any coaching, consider STAFF EXCHANGE.

After the business case was established, the project had to change. The effort and schedule estimations demanded that the

team of initially ten developers, located in two sites in different countries, had to grow to sixty within one year. While one department started growing the way management expected, the other team just grew to sixteen developers within thirty months. Most developers came straight from university, and the local architect and his manager took significant time to coach them. Years later, that team was still working with a high quality and at a good pace. The other department had collapsed due to the mismatch between expectations and fulfilment, and most of the developers were fired.

Dedicated Architect

Applies to projects that have no dedicated architect and experience trouble with their architecture, either in quality of the architecture itself, in incoherent visions, or in uncovered effort.

In a development team that has an informal design and architecture process without a dedicated role assignment, the lack of a dedicated architect can cause one or more of the following problems:

- The development focus is on management goals only. A technical focus is not present, or is randomly selected by individual developers.
- Important internal system qualities that are essential to adequately manage the system are not addressed.
- Developers who take over significant parts of the architectural tasks fall behind their schedule and get low performance ratings.
- Questions concerning the architecture are not consistently answered.
- Different people address different expert developers for technical issues.

An acknowledged architect has less time available for real programming, and is potentially expensive,	…but dealing with inconsistencies and the derived system failures is even more expensive,
Small projects can come along without much effort on architecture,	…but building a large or reuseable system requires attention to issues that hardly matter in smaller systems; neglecting internal qualities can cause a system to break under its own weight.[5]
Any architect's experience and view on the software world is limited,	…but an architect has the broadest view of the developers, and he can still delegate.

[5] Like a stranded whale who chokes because of its own weight

Newly assigning an architect within a given team might cause personal conflicts,

…but each such conflict would be present anyway, and would otherwise express itself in technical inconsistencies.

Therefore, ensure that an architect's role becomes defined and assigned to a key developer. The architect becomes responsible to create a common vision of the system, ensure technical consistency, broaden the architectural view, re-balance the different forces on the architecture, and to coach the development team about the internal qualities (the "~ilities") that are essential to crafting large software systems. In turn, all developers, managers and technical leads pass their decision competency in these areas to the dedicated architect, and provide sufficient resources – namely the working time of the architect.

Most project situations can live with a developer informally taking the architects' role; especially agile development methods advocate having no explicit assignment [*Beck99*, *Agil01*]. However, a developer's slack time is not sufficient to cover all architectural tasks. You can compromise on how the role is called, but the architect needs to be able to spend significant effort without troubling his boss or career. The architect will need dedication, explicit empowerment and time to become accepted among his peers. Much less time can be devoted to "real work" such as coding, and this needs to be reflected in the project schedule and the performance review criteria.

The obvious candidate for the assignment is the informal architect. Convince your manager to establish the architect's role by indicating the risk reduction it could bring to the project, and by comparing the consequences of not having a consistent architecture against the costs of having an architect. This process will likely take some time. Try to get support from other developers in advance, especially from external contractors that happen to be around. Their opinions will be valued higher than those of employees will.

When the team has several informal architects, the one of them who is most frequently asked is the right candidate. A team of architects can also work when each member has a distinct key area. However, one person must have the final decision.

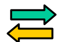
If there is no informal architect, this means that the team creates the architecture by consensus or accident, though with the best intentions. In

this case you should consider asking for an outsider to join. The same applies if there are too many architects in the team, and picking one of them would break the project.

The mechanism behind DEDICATED ARCHITECT is acknowledgement by management. Only an acknowledged architect is able to devote sufficient time, and receive sufficient respect from the team.

DEDICATED ARCHITECT involves management, the architect, and potentially all team members. The costs can become significant because you need to dedicate time to architectural issues as well as to establishing the new role in the first place. Contracted architects are even more expensive than internal ones. However, the costs for a good architect will reduce the project risk and likely pay off several times, while the costs for a bad one will lead to further cost explosion.

DEDICATED ARCHITECT has no counter indications, given that you find a capable architect. Its side effects are on the workload that the team can manage. It will decrease in the short term, but eventually increase in the mid to long term. Another side effect is the positive influence on the career of the assigned architect.



The trust in the architect is likely fostered by ↗ARCHITECT ALSO IMPLEMENTS, when the developers perceive that the architect still knows how to express ideas in code. When the team would not accept any architect, consider STAFF EXCHANGE with a significant number of people.

Staff Exchange

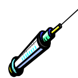
> Applies to projects stuck with old ideas that work to some extend, but lead to unsatisfying results.

In a development team that is stuck, caught within their own ideas, and blinded by their own limited experience, you need to bring in new ideas.

People's experience is valuable to the project,	…but repeated similar experiences can blind you and let you ignore new possibilities.
Changing the staff of your project is risky both socially and by means of the technical and organizational learning curve,	…but new people bring new ideas and different blinders.

Therefore, suggest to exchange some members of the development team by developers new to the domain or the company. Make sure that management replaces at least one of the key team members, and looks for replacements that are personally able to become key players within a short time. Look for developers that bring experience, a strong personality and good communication skills, so that the project really profits from their knowledge.

Ensure that you have a stake in the job interviews. Be clear about your goals and the difficult situation during the job interview, to avoid later disappointment that could counter your intentions. Management could make the first raise depending on the influence the newcomer gains during his first months.

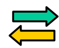


The mechanism of STAFF EXCHANGE is to influence the development team by a factor they cannot ignore: new colleagues. These bring new knowledge and different experience and inject this into the developers' minds while the entire team is going through all team building phases.

To STAFF EXCHANGE is beyond the scope of an architect and can only be suggested to management. There is no one-fits-all answer on

the related costs, but the required learning curve and the intended friction make it expensive.

A strong counter indication to STAFF EXCHANGE is when at least some of the key developers are willing to learn and accept offered opportunities. A side effect is that you run into more discussions than you really desire. Besides all irrationalities of the newly started team forming process, the arising discussions will cover development practice and methods, coding and quality standards, architectural ideas, just to mention a few. An overdose can cause the team to get lost in discussions, break its motivation, and eventually loss of the project and valuable employees. Another overdose effect could be that the company loses the expertise it once had, without being able to adequately replace it.



STAFF EXCHANGE is kind of an entropic therapy causing undirected activities. You need to complement it by problem and goal oriented therapies to focus the direction of its effect.

Before you consider exchanging the entire team, think of exchanging just the architect. An architect in the wrong place can do more harm than good. For indications of such a step, see the diagnoses in [*Marq03b*].

Another important variant is to expel the consultants. Having consultants in the role of an architect is particularly dangerous as significant competency, the reasoning behind decisions, and key knowledge will be gone at a time you cannot predict. Consultants that follow their own agenda are more an obstacle than an assistance.

A division of a consulting company specialized in technical projects was particularly good at taking the entire technical leadership of their customers projects. While about half the staff in the development team was new to that kind of projects, a large number of experienced developers were distributed over the teams. They also came to join particular projects as senior consultants when problems occurred. The internal turn-over made sure that the knowledge was spread, and that new developers became familiar with different projects quickly while maintaining a common understanding and corporate identity with the company.

"A contractor had managed to become the mind monopole at one of my customers. He motivated his queer data model with reasons about local performance gains. When the system went productive, it was a factor of 100 slower than comparable systems, which would have caused annual operation costs of several 100,000 €. Confronted with radical ideas to save a factor of 1000, the contractor reacted with denial, without being able to give reasons. Due to cost saving measures, the contractor finally had to leave the project, and system responsibility was passed to an internal team. For political reasons, only the simplest of the suggested changes became implemented, and these confirmed the initial performance gain estimation."

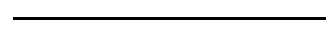
Appendix: Pattern Format

Diagnoses and therapies follow their own pattern format, which includes additions specific to the medical metaphor.

Diagnoses explore the problem at hand and give means to clearly identify what is going on and why. They also build a bridge across a variety of therapies, explain their applicability and link them to a treatment scheme.

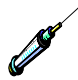
Their description starts with a small summary and a picture. The symptoms are described in depth and finished by a symptoms checklist and the diagnosis name. An introduction of the possible pathogens and the etiology closes the diagnosis' description.

Each diagnosis comes with a brief explanation of applicable therapies and how each of them works. This includes possible therapy combination and the kind of effect, curative, palliative or preventive. Finally, some treatment schemes combining several therapies are described, that serve as a suggested starting point for adaptation to the local situation at hand.



Therapies are measures, processes, sometimes medications you can apply to one or several different diagnoses. Their description is closer to a canonical pattern form, including problem, forces, solution, implementation hints and an example or project report. Their context is kept rather broad, as they are potentially applicable to a number of diseases or diagnoses.

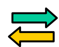
In addition to the common pattern elements, therapeutic measures contain additional sections containing the medical information. These are introduced by symbols:



The mechanisms of the therapy, why it works on which kinds of diseases

Involved roles, and costs

Counter indications, side effects, overdose effects



Cross effects with other therapies

Acknowledgements

The first and foremost thanks go to Jens Coldewey, the shepherd of this submission. His remarks helped to outline the patterns and keep them consistent, to refine the usage of the medical metaphor, and to straighten many details of individual patterns.

The workshop participants at EuroPLoP gave valuable and encouraging feedback. Thanks go to Frank Buschmann, Arno Haase, Kevlin Henney, Wolfgang Herzner, Michael Kircher, Alan O'Callaghan, Kristian Sörensen, Markus Völter, Nicola Vota, and Tim Wellhausen.

Further thanks to many unnamed colleagues who sometimes unintentional contributed to this diagnosis, its therapies and the examples.

References

Referenced diagnosis, therapy, or pattern		Referenced diagnosis, therapy, or pattern	
ARCHITECT ALSO IMPLEMENTS	*Copl95, Marq02d*	DIVIDE ET IMPERA	*Marq02d*
BIG PICTURE ARCHITECTURE	*Marq02d*	EXPLICIT DEPENDENCY MANAGEMENT	*Marq02d*
DEFINED NEGLECTION LEVEL	*Marq02d*	JOINT DESIGN	*Marq02b*
DESIGN BY SPLINTER	*Marq01a*	MENTOR	*Marq02b*
DESIGN REVIEW	*Marq01a*		

Agil01 http://www.agilemanifesto.org
"The best architectures […] emerge from self-organizing teams"

Beck99 Kent Beck: Extreme Programming Explained: Embrace Change. Addison-Wesley 1999

Cock98 Alistair Cockburn: Surviving Object-Oriented Projects. Addison-Wesley 1998

Copl95 James Coplien: A Generative Development-Process Pattern Language. In: Pattern Language of Program Design, Addison-Wesley 1995

Fowl99 Martin Fowler: Refactoring. Addison-Wesley 1999

Lipp04 Martin Lippert, Stefan Roock: Refactorings in großen Software Projekten. To be published by dpunkt, 2004 (in German)

Marq99 Dr. Kerstin Marquardt, private communication

Marq01a Klaus Marquardt: Dependency Structures. Architectural Diagnoses and Therapies. In: Proceedings of EuroPLoP 2001

Marq01b Klaus Marquardt et al: Performance Pattern Language. Report of Focus Group. In: Proceedings of EuroPLoP 2001

Marq02a Klaus Marquardt: Architecture and Organizations: Structure, Problems, and Solutions. In: Proceedings of EuroPLoP 2002

Marq02b Klaus Marquardt: Supporting the Software Architect: Selected Patterns Covering Different Perspectives. In: Proceedings of EuroPLoP 2002

Marq02c Klaus Marquardt: Principles of Performance Tuning. In: Proceedings of EuroPLoP 2002

Marq02d Klaus Marquardt: Patterns for the Practicing Software Architect. In: Proceedings of VikingPLoP 2002

Marq03b Klaus Marquardt: Neglected Architecture. In: Proceedings of VikingPLoP 2003

Trab03 Picture available at http://www.tuning-scene-droyssig.de/601maik1.jpg

Wein92 Jerry Weinberg: Software Quality Management Series. First-Order Measurement. Dorset House 1992