

Overthreading

Klaus Marquardt
Dorothea-Erxleben-Straße 78
23562 Lübeck
Germany
Email: pattern@kmarquardt.de

The classic school of software development suggests splitting the system into functional blocks. The blocks get a defined set of responsibilities, and their boundaries become clearly described. Afterwards, each function is assigned to be implemented in happy separation.

Both in embedded systems and in a world of service orientation, a system of many not-so-independent functional blocks is easily expressed as a system of many independent execution contexts (aka threads or tasks) that exchange signals. An increasing number of threads, signals, and functional blocks introduce a complexity to the system that needs to be managed.

A note to the workshop participants

Primarily I seek feedback on the completeness of the therapies, and on the precision in defining the diagnosis.

Several therapies, `DEPLOYMENT ARBITRATION`, `PHYSICAL CONTEXT AT INTEGRATION`, and `IN-PROCESS UNIT TESTS` are only provided as patlets here. Depending on your feedback, they may evolve into full patterns for the proceedings version.

An explanation of the pattern format this paper uses is appended for your information.

Introduction

While most of the following describes the development of embedded systems (or service oriented architectures), stay tuned in case you are building service oriented architectures (or embedded systems) – there will be more similarities than you might have expected.

The greatest similarity is that both are typically architected along physical boundaries and components. The common mindset is that this should be the first-order decomposition.

Separation into different physical components has many advantages. First of all, it resembles engineering along the lines of tangible things, servers or processors, physical components that can be shown to the outside world. Developers perceive their functional block as a complete entity: that it can be executed shows completion of implementation. Testing on development level can be done without much harness or code intrusion, just by running the functional block and checking whether it reacts to external events.

Taking a look at UML and the standard volumes of object oriented development, they focus on logical components of a fine granularity. This approach to system development is not invalid, but it neglects levels of abstraction and granularities that arise with complex systems. This gap could be filled with logical components of a larger granularity, e.g. components or subsystems. It may also be filled, as is the case here, with larger components that are physical and comprise both software and hardware, respectively executable programs. UML is not even very clear itself in this distinction.

Actually the mindset to combine physical and logical worlds has proven successful in many projects. Many systems require this as their central quality. However, all systems need to integrate functionality across these physical – and thus logical – boundaries. The effort to link different processes logically is significant, especially when this link was considered an afterthought.

Aside, embedded systems suffer from an additional verbal and mental twist, the two meanings of “task”. When some developer is assigned the “task” to implement some functionality, the immediate reaction is to spawn a new “task”.

This paper explores the consequences of a system design that cuts logical lines where the physical lines are. It shows how to compensate for the potential drawbacks and gain the best of different views on the system.

Diagnosis: Overthreading

Systems that are decomposed for seemingly independent components, that mistake the absence of immediate physical coupling for independence, that develop an accidental complexity where finally no individual component could be removed or changed without breaking the system.

some great figure to be added.

The fictitious project's software development starts out fairly simple. The entire functionality is decomposed into functional blocks with inputs and outputs, each block is assigned to a developer or a small team – implementation is their task.

The architecture supports decoupling of functional development. Such a task will be implemented as a distinct physical component – in SOP this will be a service, in embedded systems a thread or task. This way the component's development is decoupled from other components. Each team can finish their portion independently in the fashion they prefer, and any outside circumstances like delay, lack of resources etc will not influence other components' completion. Even mediocre quality of some component will not directly influence the system, as it could be replaced some time late in the project.

The functional blocks can be tested within a test harness providing messages with incoming data, and checking for the expected output reaction. System integration finally puts all the components together and starts the runtime system.

To everybody's surprise, things become tough – not exactly at the first integration of only a few components, but as the project progresses. It starts out with the difficulties of real projects: the integration tests are delayed due to late deliveries of individual components. Some chains of functionality can not be completed, so the partial integration becomes meaningless: it does not provide more information than the individual tests. But then, when finally all components come together, it takes significantly higher effort than expected to get the system up and running. Especially the start-up

scenarios have not been explored in advance in full detail. Many effects depends on the order of initialization, some seem arbitrary. Designers of many involved functional blocks need to discuss details of the internal behavior in order to get the startup right. When it comes to testing system behavior, some work flow routines work fine while others are dysfunctional, for no immediately apparent reason.

Following the execution paths of the missing functionalities, two key effects become apparent. The first one is that the system is hard to start or deploy. The blocks are not invoked in the exact order, or at the time they should be started. After some tuning, and many turns on the screws of thread priorities, the functions come life one by one. The responsiveness is still not what was expected, and in embedded systems you will also observe a high load.

The second effect is what always happens: the initial understanding of the system was insufficient, and the decomposition has some flaws that need to be corrected. Some of the components need revision, additional components are created, and additional signals and messages are defined.

Integrating the revised blocks turns out to be troublesome, as the integrated system is already tuned and changes to the runtime system cause portions of the fragile equilibrium to break. The expected messages do not come in time, so some tasks wait longer than expected and quit proper operation.

Such a system based on run time tuning is hard to debug: the messages and task invocations can not be observed without disturbing the system itself. To enable debugging, the code becomes instrumented with trace and logging facilities, so that the control flow can be retrieved in retrospect.

The integration takes longer with the increase of functionality. Late in the project, the last functional blocks take the most time. The overall performance degrades, but worse it is hard to predict due to the cross effects from physical execution of many components. The system does not scale with the need for task switching and the amount of messages between tasks.

When finally all functions are available, the system is susceptible to error conditions and typically needs a full restart in any faulty situation. Furthermore, the team is resistant to do any changes: “Never change a running system”. In terms of physics, the system has reached a state of an instable equilibrium, where minimal changes have large impact.

Typical symptoms:

- The system architecture separates logical functions into distinct physical components with distinct execution contexts or services.
- Physical decoupling is considered the key element to the system architecture.
- Messages are used for logical functions crossing physically decoupled components.
- Tasks for developers are mapped into blocks for independent execution.
- The initial system decomposition is considered complete prior to start of implementation.
- Testing of logical function includes the physical execution context
- The team is afraid of late changes.

Occasional symptoms:

- Integration starts late in the project.
- The responsibilities for development and integration are separated.

The pathogen is a combination of believe in the ability to design the entire system up-front, and missing mental separation of a logical component's responsibility and its physical execution.

The first belief is an expression of naivete, the human tendency to solve a problem with the mental tools learned from previous experience. This is a good thing unless the new problem turns out significantly more complex, in which situation the tools do not suffice to manage that complexity. More so, the complexity often cannot be identified by the project participants.

Overthreading (or Over-Servicing) exists when this mental misconception combines with a technical approach that makes it hard to react and change the strategy.

Box: situations when this separation is useful and strongly indicated

Therapy Overview

Ideally, development teams build the right thing the right way. Overthreaded system implement an architecture that is insufficient in the long turn. The therapies go into three different directions: avoiding the conceptional shortcomings in systems thinking and improving the architecture, helping a good architecture emerging by process, and soothing some of the pain related to mediocre technical choices.

VISIBLE QUALITIES help to become self-aware which qualities are essential for the system in the current state of development. The missing quality is best expressed in the design principle SEPARATE APPLICATION FROM INFRASTRUCTURE which assists refactoring, and in this case eases the change of focus of the architecture. A dedicated DEPLOYMENT TEAM OR DISTRIBUTION TEAM can help to establish this separation, especially when they enforce DEPLOYMENT ARBITRATION.

INTEGRATION-DRIVEN ARCHITECTURE avoids that much time is spent in individual components without an early proof of concept. Based on this, the deployment team can practice PHYSICAL CONTEXT AT INTEGRATION which prevents artificial or accidental OVERTHREADING, while still maintaining the option for strong physical separation when needed. A more pragmatic approach is to change the mindset of the development with a TRANSPOSED ORGANIZATION, in long lasting projects even more than once.

A different approach to testing can also help to SEPARATE APPLICATION FROM INFRASTRUCTURE. TEST-FIRST DESIGN helps to prevent structures that exclude themselves from testing, and thus integration. It is best established with IN-PROCESS UNIT TESTS that are in sync with the DEPLOYMENT ARBITRATION.

Visible Qualities

Applies to projects whose team focuses all work and thoughts on a few essential ideas, but ignores any 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 extent 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.

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 \nearrow 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, \nearrow 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.”



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 neglect 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**.



OVERTHREADING: Apply **VISIBLE QUALITIES** at any time during the project; early is better. They can help to learn which are the intrinsic qualities needed, and which are introduced by superstition.

- ☺ The project spends less time on ill-perceived priorities.
- ☹ Priorities and qualities do not automagically change when still the same people define them.

Separate Application from Infrastructure

Applies to projects that are not trivial in scope and team size.

Projects that define an architecture along technological lines run the risk that they bind themselves tightly to the infrastructure and become legacy code prior to their first release.

The technical infrastructure makes for a quick description and classification of a project,

...but the application domain and the infrastructure domain have different and independent life cycles.

Completing an assignment both from the domain and the technology aspects is compelling,

...but coupling them tightly prevents exchange or independent refactoring of these aspects.

Therefore, clearly separate application knowledge from technical knowledge, both in code and design. Application domain classes shall not depend on infrastructure such as operating system, database, service invocation, threads or event queues. Vice versa, infrastructure code and middleware shall not depend on application knowledge and domain classes.

Consider all functional blocks as logical components only, and exclude all considerations of physical deployment and invocation.

Organic software development would not invest effort in software infrastructure without a clear need on the business and application side. However, keeping the logical blocks separated has both mid term and long term benefits. In the mid term, the application development can proceed using test stubs that reduce the effort for functional tests. Both can be developed in parallel, and the necessary expertise serves as a natural team structuring element. There is less of a learning curve required, and less quality compromises necessary that arise from incomplete understanding. In the long term, the technology could be replaced, and the application updated without a need for synchronization.

However, keeping the separation in place requires effort that does not pay off in small teams and in projects that only last a few months. With every increment and mile stone, both aspects need to integrate and proof achievements that do not come automatically. Even in large projects there will be little effort reduction, the separation pays off because of the

circumvented learning curve and the independent evolution of infrastructure and application.



SEPARATE APPLICATION FROM INFRASTRUCTURE is effective through the related engineering and management discipline.



The work and initial costs are with the architect who needs to minimize the dependencies between both aspects and possibly teams. It requires full management support. In the mid term, before the project is close to shipment, the effort required is comparable to the effort that would be needed for education and more expensive testing. In the long term it pays off through decoupled evolution of independent packages.



Counter indications to SEPARATE APPLICATION FROM INFRASTRUCTURE are a small team (~ six developers or less) and a short expected project duration (several weeks or months). In these the initial costs to establish the necessary interfaces and enforce a technical boundary – which is not even immediately useful to project or customer – are higher than an attitude to early delivery would be, even if the costs of refactoring due to the chosen complexity are considered. Use the drive for discussion to come to an adequate dosage.



To compensate for the risks associated of strict separation, combine SEPARATE APPLICATION FROM INFRASTRUCTURE with an INTEGRATION DRIVEN ARCHITECTURE. Projects with a strict separation of concerns will also benefit from a TRANSPOSED ORGANIZATION to shift the focus to immediate customer needs.



OVERTHREADING: Apply SEPARATE APPLICATION FROM INFRASTRUCTURE at the very beginning to establish a counterpart to the deployment driven mindset.

Deployment Team

also known as: Distribution Team

Applies to projects that need to consider the software deployment.

Deployment and distribution are non-trivial arts of software engineering. It needs to be adapted several times during the project, and potentially with each customer specific installation.

It is convenient to place some deployment decisions into each developers hands,

... but the accidental complexity likely prevents effective changes or adaptations.

Project subteams are typically formed around technical expertise or workflow scenarios,

... but the technical workflows necessary after completion of the software development need special attention.

Therefore, define an explicit responsibility for deployment and distribution, and place it in the hands of a distinct team.

Let this team work in parallel to application driven teams, not to teams tailored along the technical layers. The `DEPLOYMENT TEAM` is the foremost team to take care of the application workflows that need to be addressed during production, installation, service, and maintenance.

Indicate the need for such a team early in the project, and establish its leader, possibly the software architect. Allow the team to take care that all application functionality can actually be deployed at their command.

As a variant, have the members of the deployment team rotate (similar to `ROTATE INTEGRATOR ROLE`). This increases the understanding of all team members for the difficulties and relevance of deployment.



The main mechanism of the `DEPLOYMENT TEAM` is the awareness that is shifted to the late phases of software development. In combination, deployment is removed from the early phases and from accidental attention.



Deployment Team requires management action, typically initiated by the architect. There are no extra costs related, though some of the

true software costs might become more visible to the organization, or shifted to a different department.



There are no known counter indications. An overdose is unlikely to occur, given that deployment is actually relevant to the project.



The Deployment Team may want to apply `DEPLOYMENT ARBITRATION` to foster the code's ability to adapt to different deployment solutions.

Deployment Arbitration

DEPLOYMENT ARBITRATION is the practice to arbitrarily change the deployment or distribution in unforeseen ways. This shall ensure that all code is ready to be executed anywhere, i.e. that it fulfills the criteria of “location transparency”.

Physical Context at Integration

PHYSICAL CONTEXT AT INTEGRATION is the practice to provide the context for execution or deployment, so that individual developers do not need to take care of this. In embedded systems this could imply that a small set of predefined tasks with different priorities and time schedules is prepared, and that each piece of code indicates into which of these tasks it should be integrated.

In-Process Unit Test

IN-PROCESS UNIT TEST is the common practice of embedding the code under test into the environment provided by a unit test harness.

The less satisfactory alternative is to implement a test specifically build for some component, which likely will presume some execution model or deployment assumption that we would like to defer.

Integration-Driven Architecture¹

Consider a project that comprises a large number of deliverables, potentially contributed from different groups or suppliers.

You are responsible that the entire system works in the end. You need to be able to tell whether concepts can be made working, and whether contributions are valuable.

You need to rely on agreed intermediate results,

...but intermediate results have limited relevance for the final integration.

Each group or sub-team needs to be able to work independent of other teams,

...but the results need to operate together smoothly to make sense.

Late changes are expensive in large, dispersed projects,

...but uncorrected errors would be even more expensive.

Therefore, do not consider anything done until it is integrated, and do not consider anything plausible or conceptually solved until you know how to integrate it.

Start all activities with the end in mind. Identify what you need to have to ship a working system, and work back from this end to determine what you need to have when, in order to reach your goal. Schedule the integration so that each group contributes frequently every few weeks. This integration milestone plan needs to be accompanied by a detailed integration procedure that shares the responsibilities between the different teams.

The integration steps should comprise a mixture of user valuable functions and the necessary infrastructure. Resist the temptation to focus on common technology first; only employ the exact technical portions that are needed for the application progress.

When portions of the software are contracted out, it is valuable to make the architecture an explicit part of the legal contract. The binding architecture should not stop at the level of “EJB”, “Oracle”, or “3-tier”. Surprises during integration are by far less likely when important concepts are agreed up front, like error handling strategies or data exchange sequences. In case you are unable to specify all concepts in advance, establish cooperative contracts

¹ Early version published as INTEGRATION FIRST ARCHITECTURE in [Marquardt2005]

that enable you to define them as you go. Most likely this will not be possible with a fixed price, fixed scope contract.

When different teams work towards a common and unified product, it can be helpful to introduce a steady rhythm and synchronize the integration schedules of all associated sub-projects. Every few weeks each team from the entire project delegates one or two developers to the “integration days” where the entire functionality is being set together.



The main mechanism of INTEGRATION-DRIVEN ARCHITECTURE is a limitation of the overall risk, through the ability to detect conceptual clashes and implementation insufficiencies as early as possible.



All roles in a project would be involved. The effort depends on the overall team size and development process, but typically pays off quickly due to risk reduction. As a rule of thumb, in large projects you can expect the integration effort to exceed the effort spent on initial development of individual contributions.



No counter indications are known: INTEGRATION-DRIVEN ARCHITECTURE works even for small teams provided you have a simple enough process.



The efficiency can be improved when combined with process related therapies such as ↗TIME-BOXED RELEASES and ↗APPLICATION DRIVES PROJECT.



OVERTHREADING: Apply INTEGRATION-DRIVEN ARCHITECTURE at the beginning of the project. Applied on an organizational scale, it ensures that the components under development are integrateable, but it does not change the priorities of the architecture.

- ☹ The project spends little time without visible progress.
- ☹ The process allows for explicit warning signals.
- ☹ Early success cannot be extrapolated into a reliable schedule.
- ☹ Different subprojects are coupled tighter than they expect – but not tighter than they actually were without INTEGRATION-DRIVEN ARCHITECTURE, however implicit.

Transposed Organization

Apply to projects that last for a year or longer, and that are implemented by a team of more than a dozen developers.

A software project team that is structured into several sub-teams, the distribution of team responsibilities can only follow one possible view on the system decomposition. Each project must satisfy a number of different aspects and cover a multi dimensional decomposition.

The organization into sub-teams enables a focused work,

...but each project needs to have different foci, and priorities change over time.

Different foci could be supported by having a multi dimensional team structure,

...but reporting to different leads obtains more overhead than even most large projects can afford.

Changes cause friction in reestablishing working teams,

...but important goals need to be reflected in the organization to get significant attention.

Therefore, change your project organization occasionally during the projects course so that it reflects the highest risk.

Dividing the project team into sub-teams according to functional components or layers is a very natural thing for architects to suggest, and can be highly effective in technical domains. Dividing the project team according to user visible function and workflow enables the team to deliver quickly what the user expects. All significant systems need to cover both views, but the organization cannot reflect both at the same time (Conways Law [CoHa04]).

A deliberate change in the organization forces all project participants to think in multiple dimensions, and the implementation and architecture follows the organization with a delay, a phase shift in time. The expectation of repeated reversion gets the participants used to multilateral thinking.

Such a deliberate violation of Conway's Law is always temporary as the structure of the architecture will follow after some time. You will make mid-term progress in the area of the highest risk which can easily trade off the restructuring costs.



The main mechanism is change, resulting in reactions and further changes. Different aspects are addressed in the most effective way, by changing the organization.



TRANSPosed ORGANIZATION requires management decision and can only be suggested by an architect. Its costs are similar to other costs caused by change and should be seen as an insurance fee as in ↗ EXPLICIT RISKS.



TRANSPosed ORGANIZATION has a number of side effects including communication changes, irritation, and tighter integration. If the risks associated are higher than the chances it is counter indicated. Overdose effects, when you change too often, are hidden communication due to fear, ineffectiveness due to uncertainty, and an increase in staff turnover.



↗ EXPLICIT RISKS can help to determine the feasibility of TRANSPosed ORGANIZATION. An alternative team structure would be ↗ TEAM PER TASK that avoids a breakup into sub-teams and forms teams for each small task. The tasks may both be technical or application bound.



OVERTHREADING: Apply TRANSPosed ORGANIZATION when other therapies have failed to change the mindset of the team. Maximum dosage is twice during the project's course. Do not apply it in the first quarter of the estimated project time.

- ☺ Drastic change prevents participants from maintaining any prevalent mindset.
- ☺ The change gives opportunity for determined corrections and risk mitigation measures.
- ☹ Change induces further change that cannot be foreseen.
- ☹ None of the negative impacts of OVERTHREADING is directly addressed.
- ☹ The friction from the change will consume project time.

The initial prototype phase ended with a team of five that did not need further substructure. When more developers joined the project team, the tasks and later the teams were split into different areas: database, GUI, network, etc. When field tests began, the workflows slightly beyond the trivial standards failed or were unstable. To overcome this deficiency, the team focus was shifted towards making the workflow operable, and the team structure was reorganized. The workflow teams were composed to include technical competence each.

Acknowledgements

Many thanks to Andy Longshaw for his insightful shepherding of this paper for EuroPLoP 2007, and to the EuroPLoP 2007 chairs, Lise and Till, for making this submission possible.

References

...

Marquardt2004 Klaus Marquardt: Platonic Schizophrenia. In: Proceedings of EuroPLoP 2004

Marquardt2005 Klaus Marquardt: Indecisive Generality. In: Proceedings of EuroPLoP 2005

...

Appendix: About this Paper

This paper aims at helping architects and other project participants to find out what is going wrong in their project and why, and what they can do about it.

For this purpose, the problem is described in a form appropriate for a medical disease, as a diagnosis. Known resolutions or measures are introduced as therapies. Similar to the medical world, a complex problem might have more than one solution, and a solution might help to solve more than one particular problem. Diagnoses and therapies do not stand on their own but are cross-linked back and forth.






Following this medical metaphor brings some unique features. A wealth of vocabulary becomes accessible. The metaphor also shows the limitations we face: none of the presented solutions might actually cure a particular system. Some therapies are only effective when applied preventively, others are merely palliative or might at best lead to a remission.

In this paper, diagnosis names are written in UNDERLINED SMALL CAPITALS and therapy names in SMALL CAPITALS. Names used but not listed herein are marked (↗) and can be found in the references. Both diagnoses and therapies follow their own pattern formats including sections that contribute to the medical metaphor.

- The description of a diagnosis starts with a small summary and a picture. Symptoms and examination are discussed and concluded by a checklist. A description of possible pathogens and the etiology closes the diagnosis.

Each diagnosis comes with a brief explanation of applicable therapies. This includes possible therapy combinations and the kind of effect: curative, palliative or preventive. Where available, treatment schemes are described that combine several therapies. These are suggested starting points for a successful treatment of the actual situation.

- Therapies are measures, processes or other medications applicable to one or several diagnoses. Their description includes problem, forces, solution, implementation hints and an example or project report. Their initial context is kept rather broad. For each applicable diagnosis, applicability and particular consequences are evaluated.

In addition to the common pattern elements, therapeutic measures contain additional sections containing the medical information. These are introduced by symbols and show the mechanisms of a therapy and how it works (), the involved roles and related costs (), counter indications, side and overdose effects (), and cross effects when combined with other therapies (). For the diseases it can be applied to, usage sections are added ().