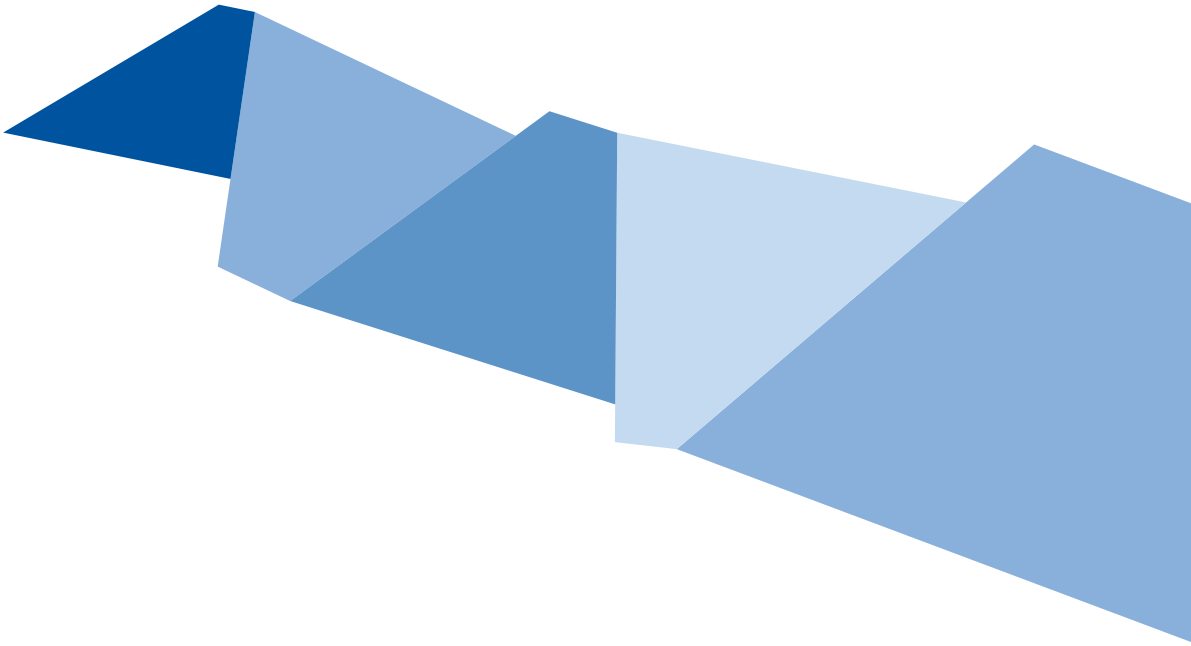

The Architecture is the Center of the Software Development Process

Manfred Nagl

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2021-08
RWTH Aachen University | Department of Computer Science | July 2021

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

The Architecture is the Center of the Software Development Process

Manfred Nagl
Software Engineering
RWTH Aachen University, 52074 Aachen, Germany

Abstract
The purpose of this paper is to explain the specific role architectures have in software development processes: The processes are only possible by extracting the essentials and denote them separately on architecture level. The architecture is the most important collection of artifacts, as the architecture influences most of the results gained in the process in a more or less deep way. The architecture not only determines the structure, the quality, and the clearness of the system to be constructed or maintained. It also determines the structure, quality, and clearness of the code, the quality assurance, the documentation, and also of the process organization.
To achieve these goals for the architecture and beyond, the architecture language has to offer corresponding clear and expressive concepts and language constructs. Furthermore, three aspects of architectures are sketched in this paper: Their role for permanent changes of the system, for even handling changes in the running development process, and for their importance to improve systems by different forms of reuse.

Key words: Architecture as essential structure for the realization of a software system, adaptability and long term properties, all development documents are determined/ influenced, the role in the process w.r.t structure, coding, quality, documentation, changes, and reuse

1 Introduction

The *architecture* of a software system /BK 03, BM 96, BR 05, GS 94, SAD, Sc 13, SEI 10, SG 96, SS 00, Wi SA/, *unites* different and important aspects, as already explained in some textbooks on architectures. We just enumerate them here and omit lengthy explanations, as they are obviously and commonly available /TP 2016/.

The *aspects are* the *following*: The architecture exposes the essential structure and hides details. It realizes all use cases and scenarios and it addresses the requirements of different stakeholders. Together, it both handles functional as well as nonfunctional requirements (restrictions and demanded quality attributes). It reduces the goal of system ownership and strengthens the organization's market position. It helps to improve the quality and functionality offered by the system. It improves external confidence in either the organization or the system. Altogether, it reduces the business risks associated with building a technical solution and builds a bridge between business and a technical solution.

Another argument for the importance of architectures is that the *architecture* of a software system has to be *known by different members* of the development team - not only by the architect(s) - at least to some extent. It is important for designers of subsystems, for programmers

of components, for quality engineers to check whether the requirements are met or to handle integration tests, for project managers to organize the group work, etc. The architecture also is the basis for discussions between different *stakeholders* of the system: developer, business manager, project manager, owner, end user, etc.

Another introductory remark corresponds to the *role of abstraction*, which is the main reason for the importance of software architectures. Nobody is able to grasp many details. The abstraction eliminates the details and extracts the essentials of a software system. There is no chance to design a system and to organize its development process without this step of abstraction and compaction. The missing details are delivered afterwards in the development process and at the right time.

A fourth remark corresponds to the *variety* of *concepts* of software architecture modelling: We find a rather *different understanding* of *architectures* /SEI 10/: (a) There is literature influenced by *practice* and industry /BK 03, BR 05, HK 07, Ru 12/, or (b) by *theory* as /GG 07, Gu 76, LZ 74, PH 12/, or (c) by programming languages having a built-in *module interconnection* language as Ada /Bu 84, Na 03/, Eiffel /Me 91, Me 97/, or (d) by patterns/ styles /BM 96, GH 95, GS 94, SG 96/, (e) *object orientation* /JC 92/, or (f) *model-based* development /BF 10/, (g) domain- driven /Ev 04/ or (h) service-oriented /KM 04/, etc. Introductory and historical papers to the field among others are /DK 76, HP 80, Pa 72/. In this paper, we rely on a more classical component/ relation approach /Na 90-20, PW 92/. This, however, is not important for the message of this paper.

The term architecture comes from *house building*. There, the essentials are also determined and planned before the physical construction starts. A similar role has the architecture of a software system. Today architects – of whatever domain they belong to – are supported by systems for document writing / retrieval and by tools helping to build up the design and construction. The role of architectures and the way of thinking in architectures have *long tradition*, shown for the design gothic cathedrals and comparing it to architectures of software engineering in /Na 19/.

The architecture of a software system should not be nice conceptual pictures as often found /So 18/, nor a sheer collection of building blocks belonging to the environment for software in a car /EN 93/, nor a DB schema without its operating application, nor a accumulation of different artifacts with more or less loose connection /BR 05/. On the contrary, it is a *precise build plan* containing all essential design decisions and a structure, by which we can study most of the problems occurring in the later realization, evaluate the main quality characteristics, and estimate the effort of the realization or of changes.

This paper is mostly on *motivation* and *explanation* level. The reader should be convinced that the architecture is the key point of software development. Technical details, as how to achieve this development, how to support it by a suitable notation or corresponding helpful tools, are not the main focus of this article.

The *contents* of this paper are as *follows*: In section 2, we introduce a model – different from usual lifecycle models – by which we explain the different tasks and resulting products occurring in software engineering, as well as the relations between these different tasks and results.

The next two sections 3 and 4 discuss the importance of software architectures by quantitative and qualitative arguments. The following two sections 5 and 6 show that architectures are the key to handle changes of a system, even dynamic situations in the development process. Two further dimensions are introduced in the next two sections: Section 7 makes clear that we have different and ordered architectures in a project, from abstract to that, which describes the shipped system. Section 8 explains the reuse dimension of different and consecutive projects, which is the main source for program families. Section 9 concludes the paper by summarizing the main ideas and naming open problems.

2 The Working Area Model and its Dependences

The *waterfall model* is well known and often criticized /Wi WM/. It orders activities according to their timely occurrence from analysis to maintenance. It is an idealization as no development follows a linear order in time. In practice, we have forward steps (thinking about future realization), backward steps (backtracking, if there was a mistake), and also iterations. The same is true for the *V-model* /Wi VM/, which goes down (detailing) from analysis to implementation and then up (integrating) to operation and maintenance. Despite of their problems, these models are widely used, not as a true mapping of reality but as simplification and idealization.

These models have the disadvantage that modeling on a certain level is done at *different places* of the *timely order*. For example, architecture activities are done in in analysis; when thinking about the future realization, in design when building up the architecture incrementally, but also in integration, installation and especially in maintenance.

Therefore, we introduce here a different model, which we call *working area model* /Na 90-20/. Its essence is to look on which *logical level* we model, and to give up the *distinction* between development, change, and maintenance, as development means more often to change or iterate than to develop straightforward. We clearly distinguish what and where we model, but also what the relations between these levels are, see fig. 1.

We distinguish between three technical working areas, namely (a) *Requirements Engineering* (RE), where we model the outside requirements and behavior, (b) *Architectural Design* (AR), where we make decisions for the realization on abstract levels, corresponding to structure and also other determinations, and (c) *Programming* in the Small or Implementation (PS), where the program code of components is developed. Furthermore, there are areas as (d) *Quality Assurance* (QA), (e) *Documentation* (DOC) (see /CB 02/ for the various aspects of documentation), and (f) *Project Organization* and Management (PO), which accompany these technical areas.

All edges between working areas of fig. 1 define a *dependency relation* in the sense that the second working area has to follow some result of the first, or in other words, its result in some way has to be consistent with the result of the first. Any of these dependency relations means something different as we see from table 2 and the following discussion.

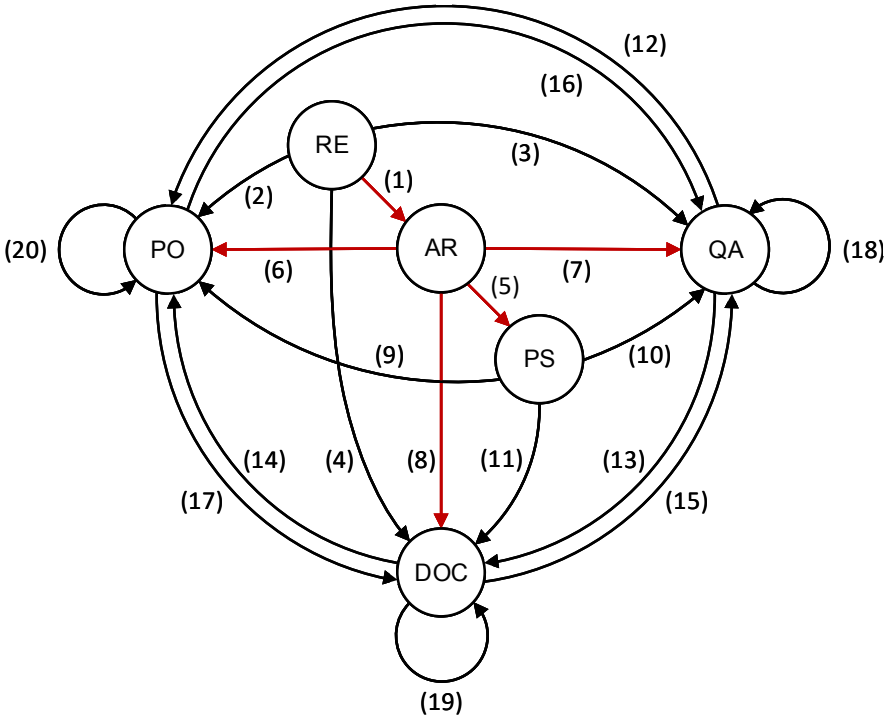

Fig. 1: Working area model and relations between working areas, in red all corresponding relations corresponding to architecture modeling /Na 90/

(1) RE - AR: Arch is a *possible structure* to *fulfil* the requirements, however Arch is not determined by the requirements
(2) RE - PO: The work packages of RE are fixed and determined, who is working for them as well as how the group of requirements engineers is built up, the total effort and the risks are determined.
(3) RE - QA: The requirements are *checked* for *completeness*, internal *consistency*, for the possibility to check, whether a *system* can be *built along* these requirements. This all is usually done by human inspections
(4) RE - DOC: The essential functional and nonfunctional requirements are described and explained in the documentation. Specific internal relations within these parts and between these parts are explained.
(5) AR - PS: The architecture *determines* the structure of the final program system, *programming* adds the *details* for the bodies
(6) AR - PO: During and after the architecture is built up, the tasks for realizing the system can be planned more precisely: Who is doing what and in which order. The total effort can be estimated, as well as the price to find out later, whether the contract will end with a positive financial result.
(7) AR - QA: The architecture is internally checked for *completeness*, *validity*, checked as a suitable structure for the requirements, or able to be *programmed*
(8) AR - DOC: The design decisions for components and relations between components, for the big parts of the architecture and their internal structure are given in the documentation.
(9) PS - PO: The programming activities have to be planned, can be estimated according to their effort, and can be checked for risks, etc.
(10) PS - QA: test by black or white box test for components, integration test in addition using the architecture, formal verification or whatever QA approach is used
(11) PS - DOC: The realization ideas of the internals of components are sketched as well as the purpose of a component in the system
(12) QA - PO: All QA activities have to be organized (planned, estimated, checked for risks)
(13) QA - DOC: The QA methods taken have to be explained and documented
(14) DOC - PO: The documentation activities have to be organized
(15) DOC - QA: The documentation has to be checked for completeness, expressiveness, and con-

sistency
(16) PO - QA: The PO activities have to be quality checked
(17) PO - DOC: The PO activities have to be explained and documented
(18) QA - QA: The quality assurance activities themselves have to be approved in the sense of quality assurance (completeness, coverage, and a suitable relation of effect and price)
(19) DOC - DOC: The specific forms chosen for documentation have to be documented.
(20) PO - PO: The work in PO has to be organized as well.

Table 2: Different consistency relations between working areas

All the above *relations are different* in their *nature,* what we now explain: From RE to AR: Construct something new consistent with the RE result, which usually is not similar; from AR to PS: construct the bodies to the components of AR and therefore extend them; from RE, AR, and PS to QA: check corresponding artifacts by using appropriate QA methods; from RE, AR, and PS to DOC: Describe the essential ideas and why they have been taken; from RE, AR, and PS to PO: estimate, plan, organize, and monitor the corresponding technical work; from QA to PO and DOC: organize and document the QA activities; from DOC to QA and PO: assure the quality and organization of documentation; from PO to QA and DOC assure the quality of PO and document the PO activities. Finally, the loops at QA, DOC, and PO mean that QA itself has to be quality assured, the different forms of documentation have to be documented, and PO has to be organized as well. We see: Any of the dependency relations means something different corresponding to the term consistency.

All the above working areas and relations are important for a software project. Especially *important* in order *to avoid* a complete *breakdown* of the project are: RE in the sense of to build the "right" system, AR to build a system "right" (namely with long-term properties like adaptability, portability and reuse), and PO for determining the costs, avoid chaos, and evaluating risks.

The graph of fig. 1 is completely *symmetric*. It allows to describe and explain *backtracking* steps in a development process. For example to go back to RE, if an error has been detected in PS which forces to correct the requirements, then going forward to architecture modeling to change according to the changed requirements. More explanation is given later. After that, we go to PS, to change the corresponding bodies and relations between components. All these steps are accompanied by corresponding QA, DOC, and PO steps.

The working area model can also be used for any *classical lifecycle model.* Even more, it facilitates *modern* life cycle *models*, as for example agile, interactive, incremental, or spiral lifecycle models, as it is clear where and what to do in a working area and what the dependency relations between different working areas mean. It helps to better understand what to do on which level and which dependences have to be regarded, as it makes activities and dependencies clearer. It also helps to keep track, if a working area is taken up again, and where to change and extend.

The working area model is applicable not only to software engineering, but also to *any engineering discipline*. These disciplines usually distinguish between conceptual design (comparable to architecture modeling of this paper) and detail design (in software engineering corresponding to implementation, here called PS). However, as these disciplines usually create

material products (which by the way contain more and more software), their life cycle also and in addition regards production preparation, the execution of the production, and later on hardware maintenance.

Software is *immaterial* and, therefore, easy to *morph*. It can be used for any application, and is often misused to correct mistakes of hardware construction. Software developers are like general engineers, because they more often change the application area and often have less knowledge or experience in and of a domain. This altogether also applies to the working area AR 'architectural design', which is the center of this paper.

3 Architecture Modelling is the Most Important Part

We start with a quantitative investigation and later go to the qualitative side.

A big Part of the Development Depends Directly on the Architecture

We go back to fig. 1 and there the technical activities and their relation to the accompanying activities and consider the *workload* of different *working areas*, see fig. 2.a. The thickness of the working areas' bubbles regards, which amount of work is spent for the corresponding working area in a software development process. Without going into details, we assume that PS and QA take about 25% of the overall amount of work. As QA is split into QA of a developer made by him/ herself and QA made externally by a member of the QA team, we get the rough figure of fig. 2.a. The same is done for documentation, which also is a common activity of the technical developers and a documentation group. By reason of simplicity, we assume that all the other remaining working areas need the same effort.

Furthermore and seen again in fig. 2.a, we look at the dependency *relations* and regard their *degree of determination* for the target working area. Initially, the relation RE to Arch is drawn by a thin arrow, as there can be quite different architectures for a system fulfilling the requirements. Therefore, there is rather little determination. On the other hand, this relation is intensively looked at in the backward direction, as the architect after essential steps always looks, whether the architecture is consistent with the requirements. Putting both arguments together, the arc from RE to AR is made a bit thicker. The relation of AR to PS is very thick, as the coarse structure of the systems' code is completely determined in the architecture. As already discussed, PS delivers the bodies of the components defined in AR. The relation AR to QA is thick, as the architecture plays a big role for QA (human inspection whether the system is consistent to RE, whether the architecture fulfills long-term properties as adaptability/ portability, traceability for changes, the architecture determines the order of components of an integration tests, etc.). This is true for the work done by the architect (internal QA) as well as that of the QA engineer (external QA). The relation AR to PO is thick, as after a preliminary architecture the PO activities become clearer and after the architecture the PO activities (estimating effort and price, workload assignment, risk calculation, etc.) have a solid ground. In a similar sense, the relation AR to DOC is important.

If we now put both arguments together, the amount of work in working areas and the degree of determinations of relations, we can conclude: *AR* does not represent the biggest workload but it *determines* by its design decisions a *big part of the overall workload*. This was the first

part of the quantitative discussion. The second statement, which we can derive from fig. 2.a, is that the architecture modelling is the *central part* of the whole development/ maintenance process.

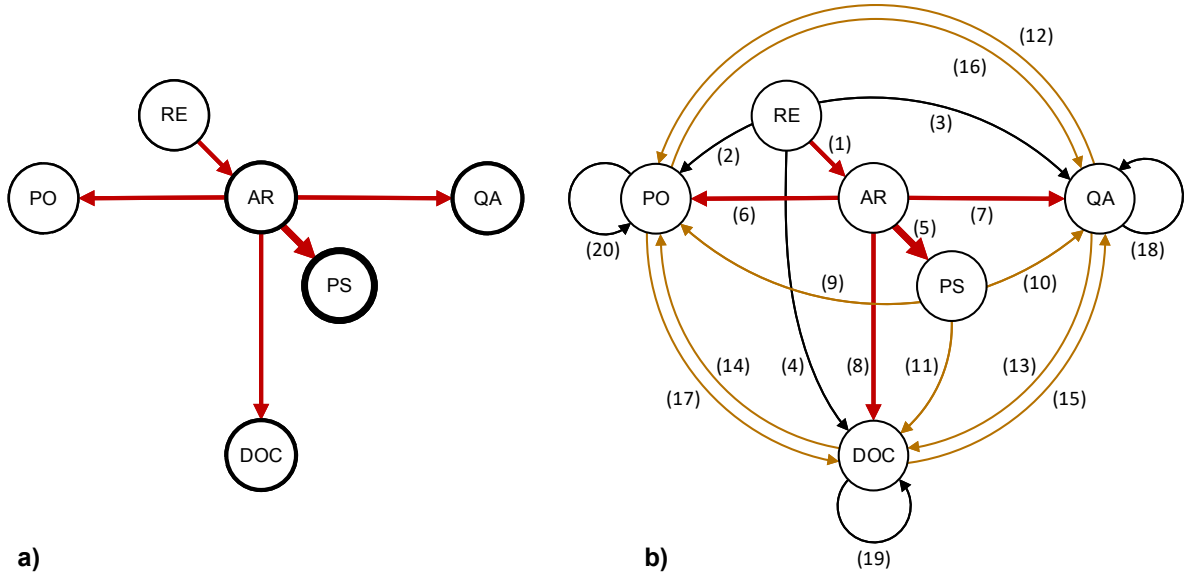

Fig. 3: (a) Workload of working areas, determination of the influence of the architecture, (b) indirect influence

Also Indirect Dependence of the Architecture

We have learned from the arguments of above that the architecture has a strong and direct influence on PS, and also a direct influence on QA, DOC, and PO. As PS, QA, and DOC also influence PO, we can conclude, that the *indirect influence of AR* on PO even emphasizes and strengthens the importance of PO, see yellow-brown relations in fig. 2.b. It even strengthens the dependency of PO on itself (PO has to be organized, which gets more complicated by these indirect dependences).

The same arguments can also be applied to get an *indirect dependency of AR on QA,* and in the same way *on DOC*.

In a smaller amount this even applies to RE, as the first step of AR is to sharply look at RE to learn the requirements. Later on, the architect permanently looks on RE in order to find out, whether the requirements are met, when building up the architecture. So, also here, we have some *indirect dependence of RE* via AR.

Summing up we can argue: The architecture has to map the requirements, but there are different ways to find the right architecture. The architecture determines the long-term properties of the program system. A good architecture also makes QA, DOC, and PO easier. A big part of the development work is more or less determined by the architecture, either directly or indirectly. This is true not only for the technical activities, but also for the activities in QA, DOC, and PO, via transitive relations.

4 And also the Most Influential Part

Above we have argued from the viewpoint of quantity. Now, we switch to a *qualitative perspective*. Let us go back to the graph model of fig. 1 and come to the question what parts are *determined* by architecture modeling, see table 4. Thereby, we can recognize that these parts are very influential.

(a) Internal Influence on design level

The architecture defines the *essential* structure (a clear abstraction from the code) of the system to be developed or to be maintained. It reflects all the essential *decisions* made on design level. The structure is complete (all components, relations, and layers).
The architecture *explains* the structure and makes it *understandable*: This part is functional, that part object-oriented, that part reflects layers of data/ detail abstractions. We can see, which parts of the system have been taken from *outside,* e.g. from a library, and which parts are developed in the running project. We also find the architecture of an underlying *old system* (in most cases not well designed), which gets new interfaces to be able to add *new parts* with a clear interface structure to the system.
We immediately detect *micro patterns* (as three components forming a model-view-controller MVC pattern), patterns for data abstraction *layers* often at the bottom, patterns for *input* or *output* structures, those for *coupling* systems, or even *global* patterns for the whole system corresponding to a structure class (as batch systems, interactive systems, or embedded systems) or an application domain (as business administration, automotive, etc.).
The architecture can be *evaluated*, e.g. for long-term internal properties, as *adaptability* (by looking on all the places where data/ detail abstraction should have been used), *portability* (by looking on all the realization changes which might hinder), reuse (by checking whether all potential places have been seen), etc.
Even more, the architecture can be evaluated, whether we have found a good *example for explanation* and *teaching*. We need good examples in a company to get some conformity for the solutions of that company, and we also need it in a lecture. Software architecture modeling does not exhaust the possibilities and potentials of good examples. Architects of houses learn from good examples. Even in software engineering or architecture books you can find examples, from which you cannot learn very much.

(b) Influence on other working areas

The architecture is the build plan for the code of the system. All the above qualitative arguments of (a) therefore also hold for the code, which is programmed according to the architecture. In the working area *programming* and coding (PS) no architecture decisions in the sense of this paper are made. The decisions made are only on the level how to structure the internals of components. The essential code quality is to a certain extent already determined on architecture level. AR modeling is restricted: If in architecture modeling we use concepts, which are not available in the programming language (e.g. OO: programming by extension, variant programming, and dispatching for an old Fortran version), we would have to extend the programming system, which is not reasonable.
The architecture determines most of the form and quality of *quality assurance*: module tests according to the white or black box approach, covering of test cases, integration tests, order of these tests, evaluating runtime traces or formal verification. The architecture also determines the organization of quality assurance, either of the work done by developers themselves, or quality assurance done by the quality assurance group.
The *documentation* should contain the explanations what has been decided, but also where and why these decisions have been made. Therefore, the quality of documentation is already mostly deter-

mined on architecture level, as there the main and essential decisions are taken. The design rationale, often commonly written by the designer and a person of the documentation group, often gets its structure 1:1 from the architecture.
Architectures and *project organization* are closely connected. This has the implication that the role of the architect is often not clearly distinguished from the role of the project organizer / manager. A clear architecture facilitates PO, and an unclear architecture hinders PO. The reason is that in a good architecture work packages are clearly defined and the relations between work packages are simple to handle, if the components' relations are loose.
The architecture has to follow the decisions made in *requirements engineering*. This is checked during and after architecture modeling. Thereby, we detect over-specification (unnecessary determination) as well as under-specification (items, which should have been specified). Especially, in the nonfunctional requirements specification we assure aspects of the process (a certain methodology to follow) as well as for the product (a certain software component to be used, the system has to run on certain hardware, etc). To a big part, his can be verified or falsified on architecture level.
Finally, we have indirect influences according to indirect relationships and even loops (both shown in yellow brown in fig. 2.b). So, we also have corresponding indirect and significant influences from the architecture side to all these working areas being the target of such a yellow-brown relationship.

Table 4: Influence of the architecture: (a) internal and (b) on other working areas

The arguments of the second part of section 3 and those of section 4 *strengthen* the statement of above that architecture modeling is the *center* of the development and the maintenance process. However, we also have further conclusions.

A good architecture is the *result* of an *intellectual challenge*, it never comes for free. In those rare areas, where everything is clear and where we have advanced reuse mechanisms (in the extreme case generating the system or parts of it from a specification), the challenge has been solved by several projects beforehand, with many good designs, by corresponding reusable structures, by tools like generators, etc. see /Na 21e/.

We need corresponding *concepts* for the notation of *architectures*, either graphical (for overviews) or textual (module interconnection language for details), which have to offer an *integrated whole* and not a variety of different and unrelated languages. We only touch this briefly in this paper, for more details see /Na 21a/. The variety is spanned by: components for functional and date abstraction, components on object and on type level, components of different granularity (modules and subsystems), relations for different purposes as local, general and object-oriented use, generic (parametrized) components, altogether in a combined language, where relations between artifacts are clear and not up to the imagination of developers.

Architectures should be able to deal with the problems of *practice*, like maintenance, reverse or reengineering, and extension. Architectures should be independent of the programming language, but we cannot map all architectural concepts to all old programming languages. The notations should be able to cover architectures for different *domains* (BA, automotive systems, etc.), and also different *structure classes* (batch, interactive, and concurrent systems). Furthermore, and as to be seen below, the architecture goes through different stages, from rather abstract to very concrete and technical. It is the *integrated view* on architectures, which makes the difference.

Software *architectures* had a great importance in our *group* in the past, in *software engineering* – and its subdisciplines as architecture modelling /Bö 94, Kl 00, Le 88, Na 90-20/, but also variant/ version control, nonstandard data base systems, authoring support, and tool construction. We also made research in different *engineering domains*, as mechanical engineering /NW 03/, process engineering /NM 08/, but also automation, embedded systems, eHome, house construction, telecom systems, ranging from construction support to reverse/ reengineering, see /Na 21a/ section 5 for a summary.

Patterns play an important *role* in *architecture* modeling. They *range* from micro patterns between few components (the mostly known form of patterns /GH 95/), over basic layers, input/ output parts, integration mechanisms for connecting different systems, to global patterns how to structure a system (multiphase compiler) out of a structure class (batch systems) and/or a domain (systems programming).

5. Architectures and Mastering the Changes

The term *architecture paradigm* describes an *idealized* situation, which never appears in practice: We build up an architecture, which contains all layers, all components of these layers of whatever architectural granularity, all relations between components, before implementation. Then we program all these components. Mostly there are *changes* of the architecture on the *way* from architectures *to implementation* of components in detail. However, the idealization is unavoidable, if we want to be able to divide labor, to check quality in advance (e.g. by human inspections), or to think in reuse before we go to realization. Thus, changes will always come up. Thereby, the architecture is changed as well.

No complex technical system is *correct*. This is not only true for software but for any complex technical system. Incorrectness means that there are errors, but we do not know, where and when they become evident. Traceability to estimate the effort of changes, and also to carry out the changes, are important. So, again, *changes* are unavoidable and necessary. This also applies to assure quality afterwards and to keep the architecture in a state that future changes are possible again.

The key to handle changes is abstraction. That means that we define interfaces, below which we forget about changes of *data* representation (data abstraction, e.g. how a complex record is built up, which files are used, which layout, style, or UI system is used, etc.) and also of a different physical, chemical, electrical, or protocol realization of an abstract *functionality* (functional abstraction) in embedded systems. In both cases, we forget about details. So, *detail abstraction* is the key to get clear and flexible architectures.

Abstraction always belongs to *layers*. The underlying form of hierarchy is not important. It might be driven by locality, general usability, or object-orientation. It might belong to a *flat* architecture (as above) or to enclosing something by a wall, i.e. the body of a component or a subsystem (*component granularity*).

The *discussion* of "*What can change*?" is very important for making architectures stable w.r.t. changes. There are two discussions: (a) Changes of the *requirements*, after having determined these requirements. This opens our mind for more general solutions on architecture level. We

do not only design an architecture for the given requirements. We also try to think about slight changes and build the architecture to cover these changes as well. A good architecture never relates 1:1 to the requirements, but it already takes care for some possible and future changes of the requirements.

The second discussion is about (b) *changes* of the *realization* after having worked out the architecture. That leads to all parts of the architecture, where we should have used detail (data or functional) abstraction. Such abstractions make the architecture stable: Changes are within the body or at most below of an interface, and they do not go up in the architecture to client components. Thus, changes stay within a *certain range*.

We discuss a simple example to clarify both changes (a) and (b) and the importance of an architecture making use of abstractions, see fig. 5. We introduce a *simple batch system* Telegram Counting System. The component Telegram_Input provides the data of different telegrams. Every telegram has a telegram identification (TID) and following words. We count the number of postal *words* (of a certain maximum length, otherwise we split; a punctuation word STOP is not counted). The *price* is determined depending on the number of postal words. Both is done by Comp_Telegram. The computed data of the telegrams are stored in Telegram_Data. The component Prepare_Print produces a paper list of the telegram data ordered by their TID. The component TD_Computation controls the program.

Fig. 5 shows the *resulting architecture*. We see functional layers on top and data abstraction layers at the bottom. Although we have data abstraction components and even abstract data types, object-orientation in the strict sense (classes and modelling commonalities and differences of these classes) plays no role in this architecture.

We first discuss *changes of the realization*, part (b) of above, to demonstrate the stability against realization changes. The Telegram_Input has two data abstraction interfaces, one for getting the data of a single telegram (get_TID, get_NextWord, Was_Last_Word, etc.), and one for the collection of telegrams (get_Next_Telegram, Was_Last Telegram, etc.). Both interfaces prevent that details of how to store a telegram or the collection of telegrams are used by the clients. The same is true for the interface of Telegram_Data. Prepare_Print uses the interface of Layout_Indep_Output, which hides all the layout and style details of the print list. All the bottom components make use of devices or files, which is only shown at the right side of fig. 5. We conclude: Any realization change has no effect up of the interface, but only below of it.

Let us now discuss *changes of the requirements*, so (a) of above. We assume an extension of the telegram system, such that the input also contains the data of the sender and the receiver of the telegram, a change of the *functionality* of the system. Thus, the interface for single telegrams is extended. We see by the import relation that Comp_Telegram can take these data. If this component does not make use of these new data, nothing further is changed. If it makes use, then the computed data are changed. These data then are usually stored in Telegram_Data, which means that the interface has to be changed. If this extended interface is not used in Prepare_Print, we are done. If they are used, the interface of Layout_indep_Output

has to be changed. We see that we can *trace within the architecture* to *discuss* and *control* the *extensions* according to changed requirements.

These discussions of above also lead us to *estimate* the changes w.r.t. time, costs, and planning of personnel.

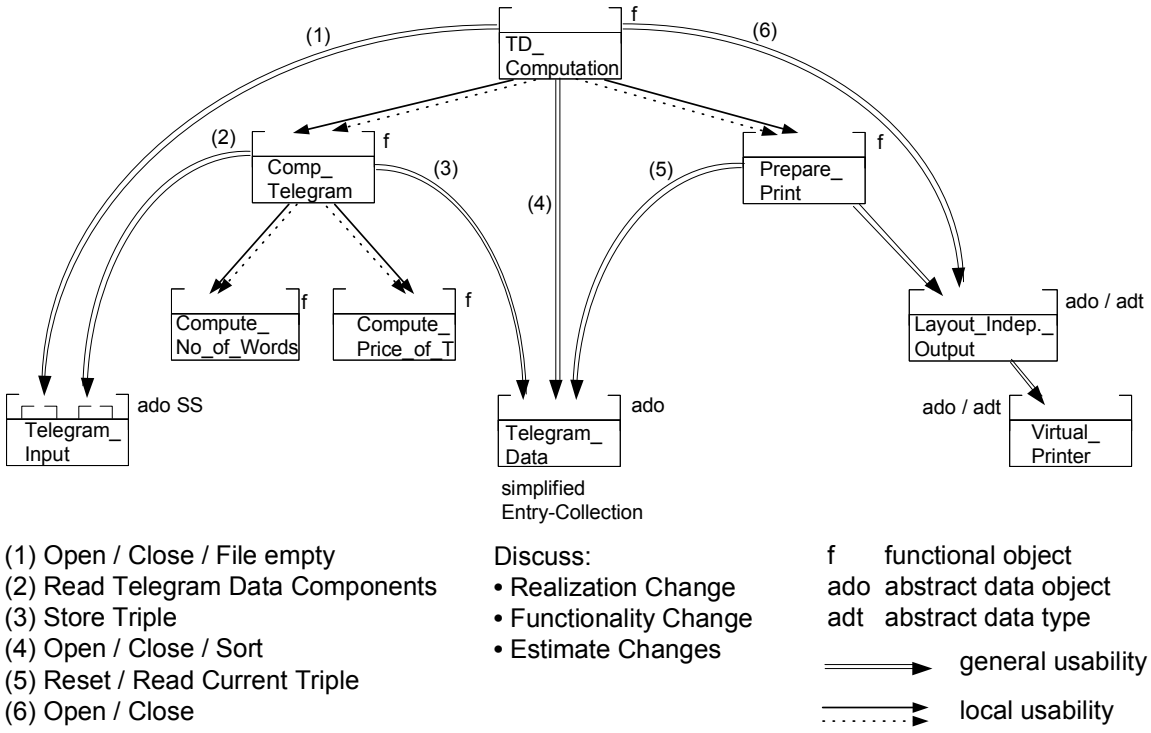

Fig. 5: (a) data abstraction makes the architecture stable, (b) traceability of interface changes

Summing up: The *organization* of the development process and, especially, of *changes* in that process is *facilitated* by a good *architecture* making use of detail abstractions. In other words, a good design has / or errors in the design have a tremendous influence not only in architecture modelling, but also on the whole development process, as already argued above.

The biggest *mistakes* are (i) to construct an architecture, which is a *1:1 mapping* of the requirements. The reason is that any change of requirements then induces a change of the architecture. The other big mistake is (ii) to ignore abstractions and *use* many *details* in the architecture. Again, in this case, any change of details implies a nontrivial and often far-reaching change of the architecture.

6 The Architecture and its Role for Dynamic Changes of the Development Process

Building up the architecture of a software system and realizing the corresponding software system is not a linear and steady activity. We see steps, which say how to go further and we face mistakes, which have to be corrected. Altogether, we cannot plan the activities in PO beforehand by a static description. We call this phenomenon the *dynamics of the development process*, as the corresponding information is only available in the running process /HH 10, HJ 08/.

The evolving plan depends primarily on the architecture. After we have designed a part of the architecture, we see how many and which components are introduced and how they are connected to each other. We see what has to be implemented, tested, integrated, documented, etc. We call this *evolution dynamics* /HJ 96/. This part of the architecture allows to define a subprocess, which now can be planned corresponding to time, money, division of labor, etc. This planning should not be done on fine-grained code level. This would be too detailed. We need the essential structure, namely the architecture.

Another situation is that when building up the architecture or even realizing the corresponding part of the software system, we detect, that there was a serious error/ mistake done before. So, we have to go back in the development history to find the place, where we have to correct. Then, there are different possibilities how to proceed forward, after having corrected: (1) the correction takes place within one component and the rest is untouched. That is the best case. (2) A part of the development process results after the mistake remains valid, other parts have to be changed. (3) The correction was so fundamental that all the results after the change have to be given up (worst case). We call such situations *backtracking dynamics*. Again, we need the architecture level to handle such cases.

A further case is that in the running development process a decision comes up to use a certain bigger component from outside. This usually has some implications around this new component (other imports, other form of usage of this component). Let s call this case *replacement dynamics*. The situation is first covered on architecture level.

So, the architecture is the right level to study these different forms of dynamics. We need a description of the tasks, which *corresponds to the management level*: What to do, who is doing it, connection to other tasks, collecting of activities for a group of developers. The architecture has the necessary degree of granularity. On requirements (RE) level, we do not know enough for the organization of the realization, on the level of programming (PS) we have too many details.

There are *further forms* of dynamics to be sketched below.

7 Different Forms of Architectures in a Project

Up to now, we only discussed one architecture, which is usually called *conceptual architecture*. It is the most abstract form of the architecture and, therefore, the form, which ignores most details. Starting from this conceptual architecture we go, step by step, to more concrete forms of the architecture building up different further views of the architecture. Thus, there is not only one architecture, we have to look at. Architectures appear in sequences in a development process. We only give a sketch of these different architecture forms here, for more details see /Na 21b/.

Let us explain this by an *example of an embedded system* /Na21c/. After (i) having elaborated the conceptual architecture, we determine, which parts of the system are (ii) executed concurrently (process components). Consequently and in addition, we have to assign synchronization protocols to those resource components, which can be accessed by these processes. The next step is (iii) that an embedded system has to be explicitly started and has to be explicitly shut

down. After explicit start and before explicit stop it "runs for ever". This start and stop is usually done by further control processes. In embedded systems, which e.g. control a plant, furthermore, we have to do something (iv) for emergency handling. This means that normal execution is abandoned to avoid that the plant is destroyed. We accept some (data) damage to avoid a more serious (hardware) damage. The next step is to (v) distribute and deploy the system, (vi) to introduce technical concurrency due to distribution and (vii) further transformations to improve the runtime efficiency of the system. The sequence starts with the conceptual architecture and ends with the form of the architecture corresponding to the shipped system. Every step in between adds further information.

There might be more than the 7 stages of the embedded system example sketched above. The question comes up, what is the *suitable order* of these different architectural occurrences and what should these occurrences express? What is before what and what is behind what? /Na 21b/ describes some rules for ordering the architecture sequence.

All *stages* are *handled* on the *architecture level*. Which stages occur depends on the domain of the system, the system complexity, the constraints the system has to obey, etc. In any case, we order according to rules, as e.g., that more likely changes are done after more stable situations. Summing up, we have ordered architectures according to time in one project, with the aim to minimize change effort.

All these stages determine the final system after development or maintenance. So, all these *stages* of the architecture are *important*, as well as the *transformations from stage to stage*. We need an annotation language (extending artifacts) or, connection language (to relate to other artifacts) to express the mutual relations between the different stages: concurrency, runtime behavior, semantics, distribution etc. /Na 21b/.

8 Reuse Ideas in Different Projects and Families of Systems

Reuse /Co 98, ICSR, LS 09/ is usually a *bottom-up endeavor* across different projects. The knowledge about reuse grows from project to project. The first projects profit from doing a similar task more than once, from project to project the experience and the competence is growing. We do it similar but better /PC 95/ and more productive /SZ 19/ (*shallow reuse*). The architecture runs through incremental improvement steps.

Then, we start to think about the process for a solution, what we have learned, what we can do better, how we can *reorganize the process* or even, where there is room for steps of automation. This is usually done in different projects. Again, experience and competence is growing, but now in major steps. We call this *deep reuse* /Na 21e/. Deep reuse changes the architecture such that after a series of changes the architecture looks completely different and the steps to be "manually" done shrink dramatically.

Another and even more complicated situation is, if we design, plan, and realize *families* of systems, or *product lines* /CN 07, Ja 00, PB 05, PK 09/. In this case, we do not think about one system, but about a set of systems, where the members share some similarity relation. And we try to make *use of this similarity* in the development process. This is another form of deep reuse.

These forms of deep reuse are *rare in industry*. Deep reuse demands for time, intelligent people, money, and the clear insight and commitment of the management that the advantage in the longer run is worth the spent money. The effect can be dramatic, but also the costs.

Again, we only give a sketch for an example here, see /Na 21e/ for more details. There the *construction* of a *multi-phase compiler* /AS 06, WG 84/ was studied as the running example. It started with shallow reuse, learning and doing better. Then the deep reuse steps followed: The basic layers, the global architecture of the compiler, the framework for the compiler as reusable product, such that only the specific phase components have to be developed. Learning about formalization and automation of these phases followed, and finally ended producing generators, which do the phase's translation automatically (compiler-compiler approach). The different steps and their transition can be described by process interaction diagrams (PIDs) /Na 21d/.

As argued above, reuse is usually improved from project to project. In rare cases, we also have dramatic changes within one project. An example is that we have a core component, which was planned to be developed internally. As this failed, something from outside had to be used, which demanded a different program structure. This subsequently induced a *dynamic change* of the process *due to reuse* within the development, as discussed for other examples in section 6. Another possibility is that this component is delivered by a subcontractor project, which is decided in the development process. This implies a *different form of organization and cooperation* /Jä 03, He 08, He 11/.

Knowledge about the development method and process usually grows from project to project. The same applies in in the case of *deep reuse*. So, in both cases, dynamic changes are mid and not short term. In some, usually rare cases it can happen in the running development process (a new and important component or tool is introduced, from an old system we derive a framework for the new). In these cases, we have *dynamic changes* within the development, as for the examples of above /Sc 04/.

9 Summary, Importance, and Open Problems

Summary and Lessons Learned

The essential *ideas* and *statements* of this paper are the following.

- The architecture of a software system unites different *aspects* and it has to be *known* by different members of a development team (not necessarily by the same level of deepness). It is based on the concept of *abstraction* (architecture paradigm). It has to have a clear syntax and semantics (not often formally defined). Thus it is not a "conceptual picture". Its role should be commonly accepted, as it is the case for house building.
- We introduced a "process" *model*, which is organized along *levels of modeling*, not regarding the timely order or a difference of construction and change, see fig. 1. This yields clear working areas, and also clear relations between them, see table 2. The model is symmetric and it can also be used for engineering projects.
- By looking where the main structural decisions are made, regarding the part of work being determined by these decisions, and estimating the amount of work for these parts, we

concluded that *architecture modelling* is in the *center* of the development process. This was studied corresponding to *quantity* and also *quality*.

- Development means permanent *changes* of results. The architecture is the artifact to master the changes. The changes can be determined by brainstorming discussions on two levels, *requirements* and *realization*. A good architecture handles realization changes by having used detail abstraction for all corresponding situations. Requirements changes are facilitated on architecture level by looking on changed interfaces and on traceability of use edges.
- A good architecture is never a 1:1 extension of the requirements specification, has hidden all details which can evidently be changed, and is also stable if minor requirements change.
- There are *dynamic changes* within a development process and also from project to project, i.e. in successive processes. The architecture is the key level to study these dynamic changes, as e.g. evolution dynamics or backtracking dynamics.
- There exist *different* forms of *architectures* in a project, from abstract to detailed. These architectures have to be ordered by reasons of clearness but also by reasons of minimizing the total change effort. Thus, we do not study one architecture, but different ones, their order, and their transformations from step to step.
- *Reuse* can happen in a *shallow* and in a *deep* form. The base for the latter are transformations of the architecture. Any transformation increases the value of reuse. In the case of the architectures with reuse, we regard variations according to requirements, which sustain the similarity between the systems. We speak of families of systems or product lines if we regard solutions for a set of systems. .This makes architecture modeling even harder. For reuse as well as for families/ lines, the architecture is the right level for discussions.

In this paper, we did *not study* the *transformations* between working areas, even not those, where the *architecture is* directly *involved* as, for example, the transformation from the requirements to the architecture. These transformations will probably be studied in a forthcoming paper.

The Open Problems

There is no *standard model of architectures*. Object-oriented architectures in BA applications and data flow architectures in embedded systems seem to have not very much in common. There are ideas of unifying these different architectural notations/ styles by a component- and relation-oriented approach /PB 16/ or a classical module/ subsystems and usability relations approach /Na 21a/, which both are not often used in industry. It is evident that we have no standardizations as it is for the architecture of houses, where we find ground plan, flor plans, views and perspectives from different directions, and refinements for installation, plumbing, etc. An initiative for a standard in house building has got more attention in recent times /ET 08/.

Architectures also have a central role in *all engineering disciplines*. Again, the commonalities of architectures in electrical engineering and subdisciplines as layout plans, in mechanical engineering for machines and also factories, in process engineering for plants together with

their automation and control are not worked out. We only find first ideas in that direction /NM 08, NW 99, NW 03/. There is much room for initiatives bridging the different disciplines.

In all the above domains, the architecture has a central role for the development process and even further, as pointed out for software systems in this paper. If there are different opinions about architectures in various domains. A less ambitious question could be: Is there at least some agreement which *questions the architecture* of different engineering domains *should answer* or which problems should be *solved* by using the architecture?

There is a strong link with an even growing importance for all engineering domains, namely the software part of engineering systems. Could that part be the starting point of thinking in commonalities in a deeper way? A trend of this kind has started in architecture of houses and civil engineering for some time /KN 07, Rü 07/.

In any of the above disciplines, there is furthermore room for *intelligent tools* /Na 96, NM 08/, more than painting or writing tools. We need intelligent tools, which support the work in depth by syntactical and semantical support for specific working areas. Especially w need tools by integrating the different aspect- and domain-specific artifacts. Here again, we have a huge field for future activities.

10 References

/AS 06/ A. V. Aho/ M. S. Lam/ R. Sethi/ J. D. Ullman: Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2nd ed. (2006)

/BF 10/ M. Broy/ M. Feilkas/ M. Herrmannsdoerfer/ S. Merenda/ D. Ratiu: Seamless Model-based Development: From Isolated Tools to Integrated Model Engineering Environments, Proc. IEEE, 98, 4, 526-545 (2010)

/BK 03/ L. Bass/ P. Clements et al.: Software Architecture in Practice, 2nd ed. Addison Wesley (2003), 3rd ed. Pearson (2013)

/BM 96/ F. Buschmann/ R. Meunier et al.: Pattern-oriented Software Architecture- A System of Patterns, Wiley (1996):

/Bö 94/ J. Börstler: Programming in the large: Languages, Tools, Reuse (in German), Doct. Diss. RWTH Aachen University, 205 pp. (1994)

/BR 05/ G. Booch/ J. Rumbaugh et al.: The Unified Modeling Language User Guide, Addison Wesley (2005)

/Bu 84/ R. Buhr: System Design with Ada, Prentice Hall (1984)

/CB 02/ P. Clements/ F. Bachmann et al.: Documenting Software Architectures, Views and Beyond, Addison Wesley/ Pearson Education , 485 pp. (2003)

/CN 07/ P. Clements/ L.M. Northrop: Software Product Lines: Practices and Patterns, 563 pp., 6th ed., Addison Wesley (2007)

/Co 98/ B. Coulange: Software Reuse, 293 p., Springer (1998)

/DK 76/ F. DeRemer/ H.H. Kron: Programming in the Large versus Programming in the Small, IEEE Transactions on Software Engineering, SE-2, 2, 80-86 (1976)

/EN 93/ ECMA/NIST: Reference model for frameworks of software engineering environments, ECMA TR/55, ECMA, 1993. URL: http://www.ecma-international.org/publications/files/ECMA-TR/TR-055.pdf

/ET 08/ C. Eastman/ P. Teicholz/ R. Sacks/ K. Liston: BIM Handbook. John Wiley & Sons, 2008

/Ev 04/ E. Evans: Domain-driven Design, Addison Wesley, 529 p. (2004)

/GG 07/ R. Grammes/ R. Gotzhein: Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science 4422, Springer. pp. 200–214 (2007)

/GH 95/ E. Gamma/ R. Helm/ R. Johnson/J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, 395 pp. Addison Wesley (1995)

/GS 94/ D. Garlan/ M. Shaw: An Introduction to Software Architectures, TR CMU-CS-94-166 (1994)

/Gu 76/ J. Guttag: Abstract Data Types and the Development of Data Structures, Comm. ACM 20, 6, 396-404 (1977)

/He 08/ M. Heller: Dezentrales, sichtenbasiertes Management übergreifender Entwicklungsprozesse, Diss. RWTH Aachen University, 501 pp. (2008)

/He 11/ Th Heer: Controlling Development Processes, Diss. RWTH Aachen University, 430 pp., AIB SE 10 (2011)

/HH 10/ Th. Heer/ M. Heller/ B. Westfechtel/ R. Wörzberger: Tool Support for Dynamic Development Processes, LNCS 5765, 621-654 (2010)

/HJ 96/ P. Heimann/ G. Joeris et al.: Dynamite: Dynamic Task Nets for Software Process Management, in Proc. ICSE 96, IEEE Computer Society Press, 331-341 (1996)

/HJ 08/ M. Heller/ D. Jäger et al.: An Adaptive and Reactive Management System for Project Coordination, LNCS 4970, 300-366 (2008)

/HK 07/ C. Hofmeister/ Ph. Kruchten et al: A general model of software architectural design derived from five industrial approaches, Journal of Systems and Software 80, 106-126 (2007)

/HP 80/ H.N. Habermann/ D. Perry: Well-formed System Compositions, TR CMU-CS-80-117, Carnegie-Mellon University (1980)

/ICSR/ International Conference on Software Reuse, Proc. from 1990 to 2017, see also Wikipedia ICSR

/Ja 00/ M. Jazayeri et al. (eds.): Software Architecture for Product Families, Addison Wesley (2000)

/JC 92/ I. Jacobsen/ C. Magnus et al.: Object Oriented Software Engineering. 77-79, Addison-Wesley ACM (1992)

/Jä 03/ D. Jäger: Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen, Diss. RWTH Aachen University, 260 pp., ABI 34 (2003)

/Kl 00/ P. Klein: Architecture Modelling of Distributed and Concurrent Software Systems Doct. Diss. RWTH Aachen University, 237pp. (2000)

/KM 04/ I. Kruger/ R. Mathew: Systematic development and exploration of service-oriented software architectures, 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), 177-187 (2004)

/KN 07/ B. Kraft/ M. Nagl: Visual Knowledge Specification for Conceptual Design: Definition and Tool Support, Journ. Advanced Engineering Informatics 21, 1, 67-83 (2007)

/Le 88/ C. Lewerentz: Concepts and tools for the interactive design of large software systems (in German), Doct. Diss., 179 pp. RWTH Aachen University, Informatik-Fachberichte 194, Springer (1988)

/LS 09/ R. Land/ D. Sundmark et al.: Reuse with Software Components – A Survey of Industrial State of Practice, in Edwards/ Kulczycke (Eds.): ICSR 2009, LNCS 5791, 150-159 (2009)

/LZ 74/ B. Liskow/ S. Zilles: Specification Techniques for Data Abstractions, Int. Conf. on Reliable Software, 72-87, IEEE (1975)

/Me 91/ B. Meyer: Eiffel: The Language, 300 pp. 1st ed. Prentice Hall (1991), 3rd ed. 2005

/Me 97/ B, Meyer: Object-oriented Software Construction, 2nd ed., Prentice Hall (1997)

/Na 90-20/ M. Nagl: Software Engineering- Methodological Programming in the Large (in German), 387 pp., Springer (1990), plus further extensions over the time for a lecture on Software Architectures from 1990 to 2020

/Na 96/ M. Nagl (Ed.): Building Tightly Integrated Software Development Environments - The IPSEN Approach, LNCS 1170, 709 pp., Springer, Berlin Heidelberg (1996)

/Na 03/ M. Nagl: Introduction to Ada (in German), 348 pp., Vieweg (1982), /Na 03/ 6th ed. Software Engineering and Ada (in German), 504 pp., Vieweg (2003)

/Na 19/ M. Nagl: Gothic Churches and Informatics (in German), 304 pp, Springer Vieweg (2019), see pp. 179-187

/Na 21a/ M. Nagl: An Integrative Approach for Software Architectures, Techn. Report AIB 2021-02, 26 pp., Computer Science Dpt., RWTH Aachen University (2021)

/Na 21b/ M. Nagl: Sequences of Software Architectures, Techn. Report AIB 2021-03, 16 pp., Computer Science Dept., RWTH Aachen University (2021)

/Na 21c/ M. Nagl: Embedded Systems: Simple Rules to Improve Adaptability, Techn. Report AIB 2021-04, 23 pp., Computer Science Dpt., RWTH Aachen University (2021)

/Na 21d/ M. Nagl: Process Interaction Diagrams are more than Process Chains and Transport Networks, Techn. Rep. AIB 2021-05, 18 pp., Computer Science Dpt., RWTH Aachen University (2021)

/Na 21e/ M. Nagl: Characterization of Shallow and Deep Reuse, Techn. Rep. AIB 2021-06, 17 pp., Computer Science Dpt., RWTH Aachen University (2021)

/NM 08/ M. Nagl/ W. Marquardt: Collaborative and Distributed Chemical Engineering – From Understanding to Substantial Design Process Support, IMPROVE, LNCS 4970, 851 pp., Springer (2008)

/NW 99/ M. Nagl/ B. Westfechtel (Eds.): Integration of Development Systems in Engineering Applications – Substantial Improvement of Development Processes (in German), 440 pp., Springer (1999)

/NW 03/ M. Nagl/ B. Westfechtel (Eds.): Models, Tools, and Infrastructures for the Support of Development Processes (in German), 392 pp., Wiley VCH (2003)

/Pa 72/ D. Parnas: On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM 15, 12, 1053-1058 (1972)

/PB 05/ K. Pohl/ G. Böckle et al.: Software Product Line Engineering, 467 pp., Springer (2005)

/PB 16/ K. Pohl/ M. Broy/ M. Daemkes/ H. Hönninger (Eds.): Adavanced Model-based Engineering of Embedded Systems – Extensions to the SPES 2020 Methodology, Springer, 303 pp. (2016)

/PC 95/ M.C. Paulk/ V.V. Weber/ B. Curtis/ M.B. Chrissis: The Capability Maturity Model: Guidelines for Improving the Software Process. SEI series in software engineering. Reading, Mass.: Addison-Wesley (1995)

/PH 12/ K. Pohl/ K. Hönninger/ H. Achatz/ R. Broy (Eds.): Model-based Engineering of Embedded Systems – The SPES 2020 Methodology, Springer, 304 pp. (2012)

/PK 09/ A. Polzer/ S. Kowalewski/ G. Botterweck: Applying Software Product Line Techniques in Model-based Embedded Software Engineering, Proc. MOMPES'09, 2-10 (2009)

/PW 92/ D. E. Perry / A. L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17:4 (1992).

/Ru 12/ B. Rumpe: Agile Modeling with UML (in German), 2nd ed., 372 pp. Springer (2012)

/Rü 07/ U. Rüppel: Vernetzt-kooperative Planungsprozesse im konstruktiven Ingenieurbau, Springer (2007)

/SAD/ Software Architecture and Design Tutorial, TutorialRide.com, https://www.tutorialride.com/software-architecture-and-design/software-architecture-and-design-tutorial.htm

/Sc 04/ A. Schleicher: Roundtrip Process Evolution Support in a Wide Spectrum Process Management System, Diss. RWTH Aachen University, 330 pp. (2004)

/Sc 13/ R.F. Schmidt: Software Engineering – Architecture-driven Software Development, 376 pp. Elsevier (2013)

/SEI 10/ Software Engineering Institute of CMU: What Is Your Definition of Software Architecture, https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=513807

/SG 96/ M. Shaw/ D. Garlan: Software architecture: perspectives on an emerging discipline. Prentice Hall (1996)

/So 18/ I, Sommerville: Software Engineering, 10th edition in German, Pearson (2018)

/SS 00/ D. Schmidt/ M. Stal et al.: Pattern-oriented Software Architectures, vol 2 Patterns for Concurrent and Networked Objects, Wiley (2000)

/SZ 19/ C. Sadowski/ T. Zimmermann (Eds.): Rethinking Productivity in Software Engineering, Springer Science+Business Media (2019)

/TP 2016/ Tutorials Point: Software Architecture & Design Tutorial, 74 pp, (2016)

/WG 84/ W. M. Waite/ M. Goos: Compiler Construction, Springer (1984)

/Wi WM/ Wikipedia: Waterfall Model, access April 2021

/Wi VM/ Wikipedia: V-Model (Software Development), access April 2021

/Wi SA/ Wikipedia: Software Architecture, https://en.wikipedia.org/wiki/Software_architecture

Prof. Dr.-Ing Dr. h.c. Manfred Nagl, Emeritus
Informatics Department, RWTH Aachen University
nagl@cs.rwth-aachen.de

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of (more than 570) reports dating back to 1987 is available from

`http://aib.informatik.rwth-aachen.de/`

or can be downloaded directly via

`http://aib.informatik.rwth-aachen.de/tex-files/berichte.pdf`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2018-03 Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

2018-04 Andreas Ganser: Operation-Based Model Recommenders

2018-05 Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

2018-06 Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking

2019-02 Tim Felix Lange: IC3 Software Model Checking

2019-03 Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

2019-05 Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems

2020-02 Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case

2020-03 John F. Schommer: Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken

2020-04 Gereon Kremer: Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems

2020-05 Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe: Pre-Study on the Usefulness of Difference Operators for Modeling Languages in Software Development

2021-01 Mathias Obster: Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe

2021-02 Manfred Nagl: An Integrative Approach for Software Architectures

2021-03 Manfred Nagl: Sequences of Software Architectures

2021-04 Manfred Nagl: Embedded Systems: Simple Rules to Improve Adaptability

2021-05 Manfred Nagl: Process Interaction Diagrams are more than Process Chains or Transport Networks

2021-06 Manfred Nagl: Characterization of Shallow and Deep Reuse

2021-07 Martin Schweigler: Ground Surface Pattern Recognition for Enhanced Navigation

2021-08 Manfred Nagl: The Architecture is the Center of the Software Development Process

2021-09 Manfred Nagl: Architectural Styles: Do they Need Different Notations?