PhD Theses in Experimental Software Engineering

Vol. 38

Ralf Carbon

Architecture-Centric Software Producibility Analysis

FRAUNHOFER VERLAG

PhD Theses in Experimental Software Engineering

Volume 38

Zugl.: Kaiserslautern, Univ., Diss., 2011

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

ISBN 978-3-8396-0372-7
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970-2500
Telefax +49 711 970-2508
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Architecture-Centric Software Producibility Analysis

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation
von

Dipl.-Inform. Ralf Carbon

Fraunhofer Institut für Experimentelles Software Engineering (IESE)
Kaiserslautern

Berichterstatter:	Prof. Dr. Dr. h.c. H. Dieter Rombach Prof. Dr. Ralf H. Reussner
Dekan:	Prof. Dr. Arnd Poetzsch-Heffter
Tag der wissenschaftlichen Aussprache:	16.11.2011

D 386

Acknowledgement

Since October 2002, I have the chance to work as a full researcher under the direction of Prof. Dieter Rombach in the Software Engineering community of Fraunhofer IESE and the AG Software Engineering at the University of Kaiserslautern. Thank you Dieter for this great opportunity and for your advice that enabled me to retrieve this doctoral degree. My further thanks go to Prof. Ralf Reussner for the great discussions on software architecture, especially related to producibility, and for taking over the role of my second supervisor. Thank you also to Prof. Berns for taking over the lead of my dissertation committee.

The completion of this thesis required enormous effort and motivation in the last years that I was only able to spent because of the support of many people. First of all, I want to thank my family. My wife Katrin has supported me since we know each other and especially in the last year, she and our newborn son Fabian spent many evenings and weekends without me to give me the time to write my thesis. Thank you to my parents, which supported me all the time and laid the foundation for my professional career by means of their education and by enabling me to study computer science.

Furthermore, I want to thank my friends and colleagues that accompanied me from my studies until now, namely Christian Denger, Jörg Dörr, Michael Eisenbarth, Tom König, and Marcus Trapp (in alphabetical order). Especially Marcus Trapp never got tired in motivating me to finish my thesis and his and Jörg Dörr's advice as my final internal supervisor helped me significantly. Thanks a lot also to Matthias Naab and Thorsten Keuler for the helpful discussions on software architecture and their feedback on my final presentation, and to Dirk Muthig who supported me in defining and defending the proposal of my thesis.

Abstract

Software Engineering significantly matured in the last decades, but still many projects suffer from delays, exceed their budget, do not reach their quality goals, or even fail. We experienced that many industrial projects suffer from a misalignment of software architecture and software project plan. Work activities refer to the same architectural elements causing conflicts and delays. Architectural elements are modified repeatedly in many iterations of a project causing effort overhead. Dependencies between architectural elements imply dependencies between resources causing communication overhead and delays. Other engineering disciplines like manufacturing put specific focus on aligning product design and production plan to prevent problems as mentioned above. Software Engineering so far did not specifically consider the relationship of software architecture and software project plans and did not deal with a potential misalignment of them.

In this thesis, we introduce the alignment of software architecture and software project plan as a new quality property of software called producibility and propose a method to analyze the producibility of a software product. The producibility analysis method semi-automatically detects critical architectural elements and project planning elements like work activities, iterations, or assigned resources that are supposed to cause delays or effort overhead during realization of a software product. The producibility analysis method provides recommendations to architects and project planners on how to deal with the critical elements.

The following technical contributions enable a producibility analysis: A meta-model of software production defines the relationship of software architecture and software project plans. A quality model defines producibility in a measurable way. Producibility views provide the possibility to model the relationships of architectural elements and project planning elements like work activities, iterations, or available resources explicitly in a certain project. An algorithm detects critical elements based on the producibility views and the metrics provided by the quality model of producibility. A prototype tool supports modeling the producibility views and determining the producibility metrics. Checklists support in analyzing critical elements in detail and deriving recommendations.

In an industrial case study, we identified more than 90% of critical elements up-front. We determined based on estimates of the project team that we could have saved 29% of time in the first of two iterations. This would have provided the chance to spend the saved time and effort in the second iteration, which has not been finished as planned.

Table of Contents

List of Figures

List of Tables

1 Introduction

This chapter introduces the practical and scientific problems addressed in this thesis. An overview on the solution approach is presented highlighting the main contributions of this thesis. The expected benefits are pointed out and the research hypotheses underlying this thesis are presented. The chapter ends with a summary and an outlook on the remainder of this thesis.

1.1 Practical Problem

Today, software is omnipresent in our life. We use software on our desktop computers and mobile devices many times a day. Embedded software is controlling systems and devices we regularly utilize like cars, home appliances, etc. In [BJN+06], software is called a basic material of today's innovative products. Consequently, the software industry needs to produce software on a large-scale to fulfill the huge demand for it. Thereby, production of software means to create software systems efficiently based on a well-defined software architecture and a well-defined production or project plan. Only the combination of a software architecture with a corresponding production or project plan enables software production as the combination describes what to build and how.

In hard goods manufacturing, the term production refers to a process that is highly optimized to come up efficiently with high numbers of a specific product and runs through without unplanned delays, effort overhead, and quality issues occurring. The term production is used in this thesis for software, because many software organizations specialize to certain markets and create similar products in large numbers for their customers although each product somehow appears to be individual. Even if they develop a single system from scratch for a certain customer, the project typically runs through many iterations that each should follow the same planned production process and add increment by increment to the system according to the architecture to be successful.

The rise of Software Engineering [Rom09] [Jal10] [Som10] in the last decades has led to an industrialization of software development. Large and complex systems can be developed more systematically and with less risk involved by using software architecture as a conceptual tool to structure software and control complexity, by applying well-defined software engineering processes, and by planning and managing software projects. Reuse approaches like, for instance, Product Line Engineering (PLE) [BFK+99] [CN02] enable organizations in many cases to increase the

quality of their software products, to reduce effort and time to market, and to deliver solutions customized to the individual needs of customers at a reasonable price [HOF11]. Software is developed in huge consortia of specialized suppliers and solution providers or integrators. Hence, software is developed similar to other industrial products. Nevertheless, still many software projects exceed their budget, deliver too late, do not completely fulfill their requirements, or even fail [Sta09].

Software is not produced today.

One of the major reasons for this situation we observed in many architecture assessments in industry is a misalignment of the architecture of a software system with the project plan, the development process, and the overall project organization. Hence, software is not really produced. The development process is too often bothered by delays, effort overhead, and quality problems originating from design flaws or inappropriate decisions in the earlier phases architectural design and project planning.

The software architecture of a software system and the project plan are two key artifacts for the production of software.

According to [BCK03], software architecture is defined as follows:

Definition Software Architecture: *"The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them."*

The architectural design of a software system conveys the key technical decisions taken to satisfy the requirements in a software system. In that sense it describes what to produce. It enables prediction of the quality of the resulting product, serves as a means for communication in the project, and constrains the production of the software product. Implementation and design are constrained in their creativity and are not allowed to violate the architecture. Architectural decisions, for instance, on technologies to be used, affect the processes and tools to be used to produce the product. Functional decomposition influences, for instance, release planning as it can facilitate but also hamper adding increments to a system over time (see Example in Section 1.2). According to Conway's Law [Con68], the structure of the system should match the structure of the project organization to minimize communication overhead.

According to the IEEE Standard for Software Project Management Plans [IEEE98], a software project management plan (project plan) is defined as follows.

Definition Software Project Plan: *"A Software Project (Management) Plan is the controlling document for managing a software project; it defines the technical and managerial processes necessary to develop software work products that satisfy the product requirements."*

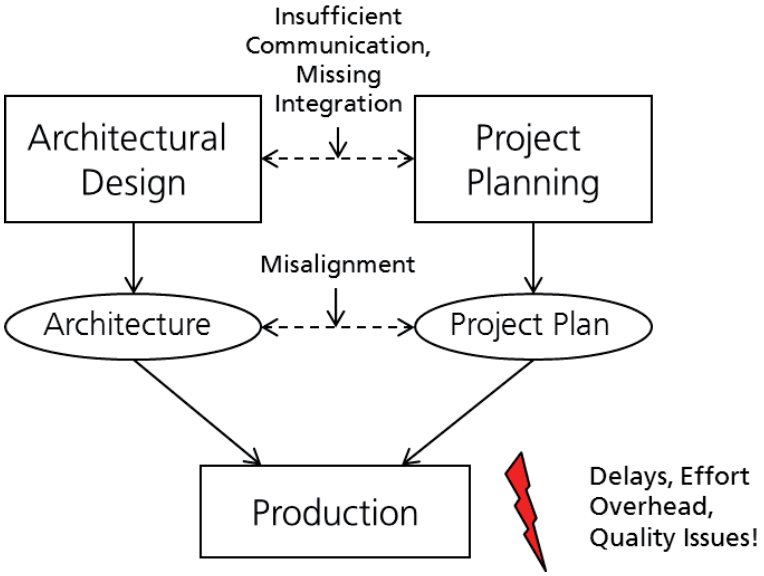


Figure 1: Practical Problem

A project plan describes how to build a software system. It contains decisions on releases, resource assignments, or schedules. It constrains the production of a software system in a sense that each organizational unit in a project knows which tasks to perform, how to perform them, and which deadlines exist. Production tasks can only be precisely defined, assigned to organizational units, and scheduled based on the architecture. The architecture defines the elements that make up the system and need to be assigned to resources. The properties of such architectural elements influence the resource assignments as not each organizational unit is equipped with the required skills to produce an architectural element. The dependencies of architectural elements dictate the required communication channels between organizational units. Size and complexity of architectural elements influence schedule and effort estimates.

Unfortunately, architectural design is often conducted without sufficiently considering, for instance, release plans, the structure of the project organization, the skills of the assigned resources, or the processes and tools adopted in an organization. Project planning is often performed without considering the architecture of a system, many times project plans are largely fixed before a first version of the architecture exists [Pau02]. Hence, architectural design and project planning are typically conducted largely independent of each other although the decisions made in each activity are heavily related.

The problem is illustrated in Figure 1. Architectural design and project planning are not sufficiently interwoven if at all. Project plans are set-up before the architecture has been designed or without considering it [Pau02]. As a result, architectures and project plans are misaligned. Problems originating from the misalignment of architecture and project plan appear during production leading to project delays, effort overhead, and

poor quality of the final product, i.e. the requirements in the product and the business goals of a project are not fulfilled in the end.

Therefore, this thesis aims at preventing such problems in practice by assuring a better alignment of architecture and project plan before production by an increased communication between architects and project managers during architectural design and project planning.

The following example illustrates the problem of misaligned architectures and project plans by considering functional decomposition as part of architectural design and release planning as part of project planning and their interrelation.

1.2 Example

During functional decomposition, architects decide on how to assign the system's functionality to architectural elements. Thereby, they apply general design principles like information hiding, try to reduce coupling between architectural elements, and increase cohesion. Project managers plan releases based on the system's functionality and constraints defined by various stakeholders involved in the project. The customer, for instance, can provide a prioritization of the system's functionality that affects release planning, i.e. the order of realizing certain features.

The functional decomposition chosen by the architect and the release plan of the project manager can be in conflict. Conflict means in this case, that a chosen functional decomposition can bear the risk of delays, effort overhead, or quality issues if the system's functionality is realized in the order defined in the release plan.

Figure 2 and Figure 3 show two alternative functional decompositions of the system. In both cases the features F1, …,F9 are assigned to architectural elements. In the first case the features are assigned to the architectural elements A1,…,A9, in the second case to the architectural elements B1,…,B9. Several architectural elements contribute to the realization of a certain feature in the given example, which is marked by a cross (X). In both alternatives of functional decomposition, three architectural elements contribute to each of the features. A 1:1 mapping between features and architectural elements is theoretically desirable but typically not realizable for all features of a system in practice. Hence, the example reflects a realistic setting in practice.

The release plan foresees to realize the system's functionality in three increments. In Increment 1, the features F1, F2, and F3 are realized, in Increment 2 F4, F5, and F6, and in Increment 3 F7, F8, and F9. Increment 1 is supposed to be finished until t1, Increment 2 until t2, and Increment 3 until t3.

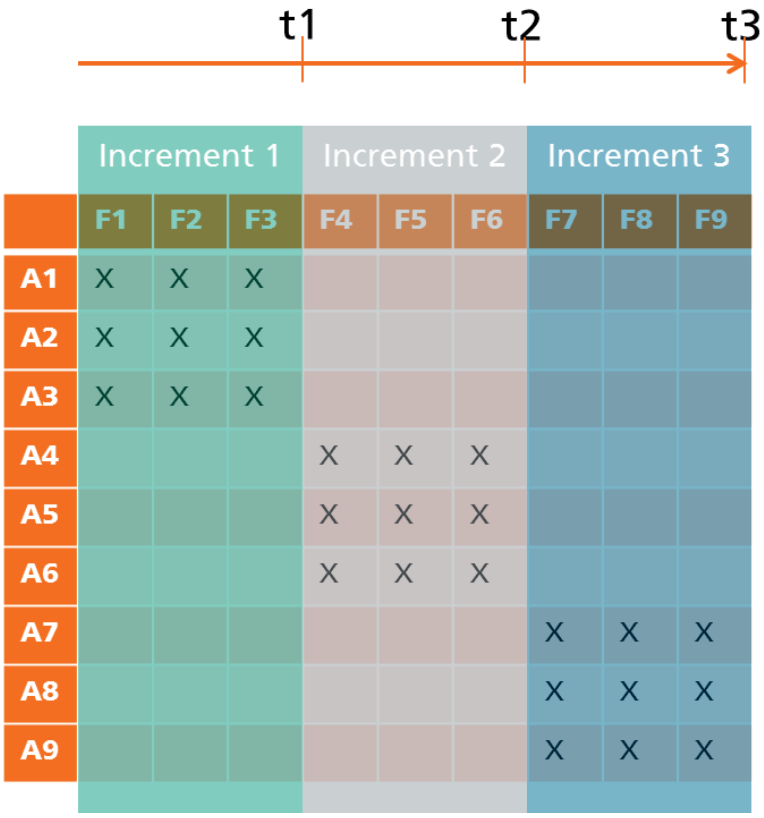


Figure 2: Functional Decomposition - Alternative 1

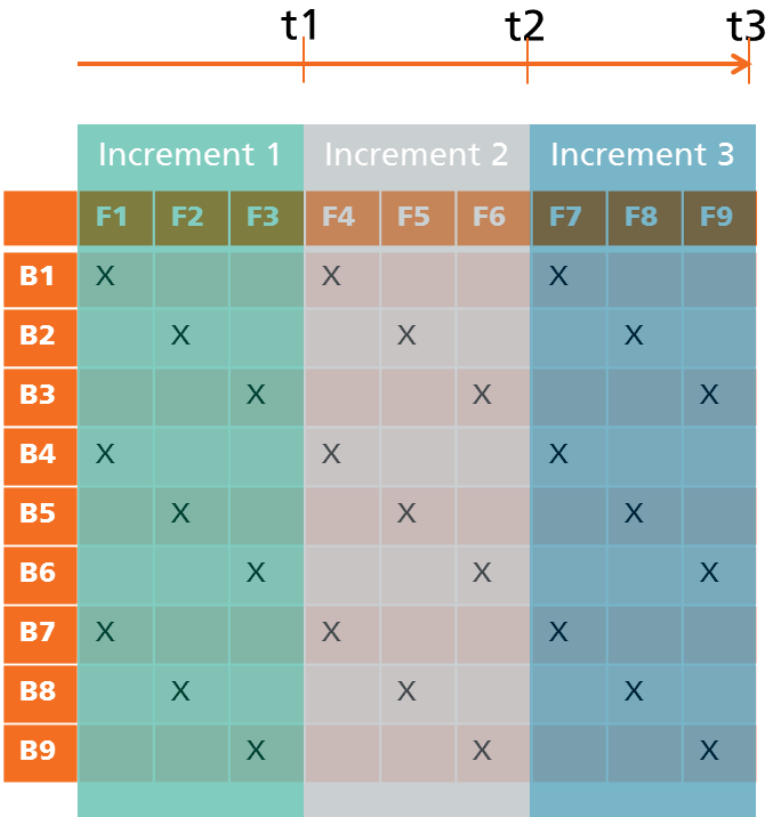


Figure 3: Functional Decomposition - Alternative 2

If the functional decomposition shown in Figure 2 is chosen, three architectural elements need to be realized in each increment. In Increment 1, for instance, the architectural elements A1, A2, and A3 need to be realized. After Increment 1, A1, A2, and A3 are completed, i.e. they do not need to be touched in later increments. As a consequence, they can also be completely tested in Increment 1 and no effort for regression testing in later increments needs to be spent, for instance.

The situation is different in the case of the functional decomposition chosen in Figure 3. In Increment 1, a first version of all architectural elements of the system needs to be realized. In Increment 2 and Increment 3 all architectural elements need to be touched again. This leads to the situation that no architectural element is finished before the end of the project. Touching all architectural elements in each increment requires effort for regression testing. Changing an architectural element often bears the risk of introducing defects, especially if different developers are responsible for the changes over time. Changes to architectural elements typically get more complex over time as the internal structure of the architectural elements deteriorates.

We can conclude from the example that the functional decomposition shown in Figure 2 should be preferred over the one shown in Figure 3 with respect to the chosen release plan. The example shows, that architectural decisions (in this case decisions on functional decomposition) should not be taken without considering project planning decisions (in this case decisions on release planning).

1.3 Scientific Problem

As mentioned above, the relationship of the architecture with the project plan is not sufficiently considered if at all during architectural design and project planning in practice. Several reasons for this situation can be identified which originate in shortcomings in the current state of the art regarding architectural design and project planning. Five closely related scientific problems (SP) that are addressed in this thesis are discussed in the following.

SP1: Missing enforcement of communication between architects and project planners during (initial) architectural design and project planning

Both, architectural design and project planning approaches do not sufficiently force or even guide architects and project planners to communicate with each other and align the architecture with the project plan.

Most architectural design approaches like Attribute-Driven Design [WBB+06] or Fraunhofer DSSA [DFK98] distinguish between the two basic activities of functional decomposition and quality-driven design. Functional decomposition leads to a basic overall structure of the system that is then revised (e.g., architectural elements are added, removed or modified) based on the quality requirements addressed during quality-driven design. Release planning is not explicitly considered during functional decomposition according to one of the architectural design approaches mentioned above. While it is up to an experienced architect to include information on release planning in architectural decision making, architects are neither explicitly forced by existing approaches to request information on release planning from project planners nor guided on how to use such information. It remains unclear, how a functional decomposition facilitating the realization of a certain release plan should look like.

Quality-driven design considers quality and business requirements as input. Typically, such requirements are specified in form of architectural scenarios. Quality requirements are related to run-time (e.g., availability, performance) and development-time properties (e.g., modifiability, testability) of the system. Business requirements, for instance, can be related to time to market, cost, or specifics of the targeted market like providing interfaces to certain systems, supporting certain communication protocols, etc. Especially addressing business requirements needs a close interaction of architects and project planners as both architectural and project planning decisions directly affect the fulfillment of such business requirements.

Most project planning approaches do not explicitly consider the architecture in early phases. Often, project plans are already fixed to a large degree including effort estimations before architectural design has started. Effort is typically estimated based on cost models like CoCoMo II [BAB+00] or by using Function Point Analysis [ISO09]. The Architecture-Centered Software Project Planning (ACSPP) Approach proposed by Paulish [Pau02] enforces to conduct high-level (architectural) design and project planning in parallel. Release planning is performed under consideration of the architecture. As a basic strategy, ACSPP proposes, for instance, to first produce a vertical slice of the system in a first release and then to incrementally add additional functionality in further releases. ACSPP also proposes to consider, for instance, the effect of global distributed development on the architecture. Effort estimates are based on the current version of the architecture and are input to scheduling a project. Furthermore, ACSPP uses the results of a global analysis [HND99] or architecture evaluation to identify project risks. However, ACSPP does not provide guidance on how to integrate architectural design and project planning activities. According to ACSPP, the project manager requests information on the architecture from the architecture team, but the flow of information back to the architecture team to challenge and potentially revise the architecture with respect to the project plan is not

specified in detail. Project management and architectural design decisions are still taken largely independent and do not explicitly mutually challenge each other early on.

SP2: No support for the identification and analysis of critical architectural or project planning elements

The identification of critical architectural and project planning elements can be seen as a first step towards a better alignment of architecture and project plans. Architectural elements can be modules, subsystems, or components. Project planning elements can be releases, milestones, or resource assignments. Thereby, an architectural or project planning element is supposed to be critical, if significant risks are related to it that can lead to production problems later on.

Critical architectural elements today can be identified by evaluating the adoption of general design principles like coupling, cohesion, or information hiding in an architecture. If an architectural element AE is highly coupled with other architectural elements, for instance, several risks are related to it that can lead to delays in completing the architectural element or in effort overhead [BCK03]. Architectural elements using AE require a working version of it before they can be completed. Such architectural elements can start working with the interface of AE specified up-front, before AE is finished. But practical experience shows, that interfaces change during production or architectural elements in the end to not behave as specified in their interface. Hence, AE is potentially critical for production as there is the risk that it gets a bottleneck during production. If AE itself depends on many other architectural elements, there is the risk that it cannot be completed in time.

But the fact that an architectural element is highly coupled does not necessarily lead to a production problem. In fact, additional context factors like, for instance, who produces the architectural element, which technology is used, how is the architectural element internally structured, etc. must be considered to decide if an architectural element is critical and if appropriate countermeasures need to be taken. Approaches explicitly supporting such a detailed analysis do not exist today.

SP3: No integrated meta-model of architectures and project plans

The basis for communication is a common language. Architects and project planners today cannot build their communication on a better alignment of architectures and project plans on a common language or meta-model. While various meta-models for software architecture or project plans exist, an integration of such meta-models is missing today. Consequently, a better alignment of architecture and project plans is compli-

cated or even prevented by the missing definition of architecture and project planning concepts in a common meta-model.

SP4: No quality model for the alignment of architecture and project plan

The alignment of the architecture with a project plan can be seen as a quality attribute of a software system because it is essential for the success of a software project. Existing software quality models like ISO 9126 [ISO01] do not include the alignment of software architectures and related project plans as a quality attribute and consequently also do not provide any guidance on how to achieve higher alignment of an architecture and a related project plan.

SP5: No integrated documentation models of architectures and project plans

Activities towards a better alignment of architecture and project plans must be based on a common documentation model of architects and project planners. According to the state of the art software architectures are documented by means of architectural viewpoints and views as well as textual descriptions documenting architectural decisions and rationales [CBB+03]. According to the general principle separation of concerns architectural views contain information on the architecture relevant for the point of view of specific stakeholders and leave out all information not relevant to the respective stakeholders and their related concerns.

Various architectural views are relevant for project planners. Structural views, for instance, support in understanding the system structure and assigning resources to architectural elements. Deployment views can be used to identify the required hardware resources and operators involved in the project. But architectural views explicitly combining architectural and project planning information are scarce. In the literature, a resource assignment view is often referenced showing the assignment of resources to architectural elements. Other architectural views combining architectural and project planning information are missing today.

Project plans are typically documented by a combination of textual descriptions, tables, and diagrams like Gantt-Charts. Visualizations combining architectural and project planning information are not existing today.

Figure 4 shows how the scientific problems are related to the overall problem context introduced in Figure 4.

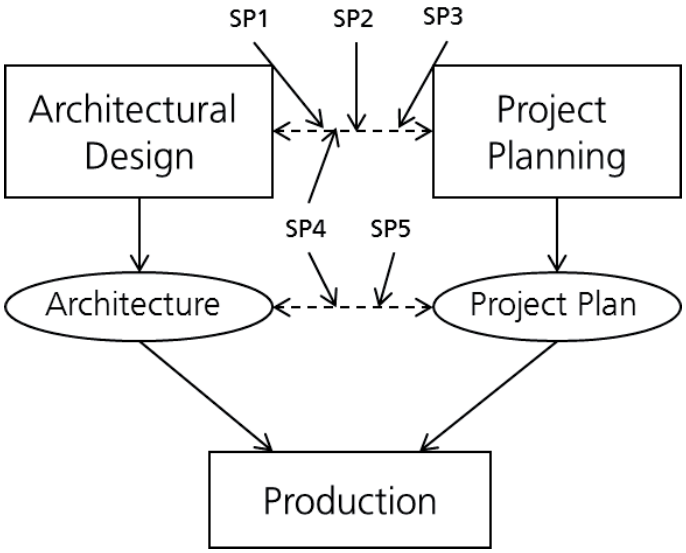


Figure 4: Scientific Problem

1.4 Solution

This section introduces the general solution proposed in this thesis. As the solution idea is partially adopted from the current state of the art/state of the practice in the manufacturing industry, the situation in the manufacturing industry is shortly described as background information.

Situation in the manufacturing industry

The manufacturing industry also suffered from problems similar to the misalignment of software architecture and project plans [GRD+97]. Products can be badly manufacturable if not designed properly. Bad manufacturability is typically measured in terms of time to market, effort, and cost. One of the identified reasons for that is a more or less strict separation of design and production planning teams [UE95]. After designers had finished their work, they handed over the design of the product to the production planning team. Typically, production planners detected problems in the design that made it hard or even impossible to manufacture in time or within budget. The detected problems were related, for instance, to the assemblability of products, the chosen materials, the available production technology, or the involved suppliers. Consequently, production planners had to ask designers to rework their solution in additional design iterations, which caused additional effort and delays already in early phases. Luckily, they at least did a thorough production planning including a check of the design with respect to its manufacturability before they started manufacturing it in large numbers.

The general solution to reduce the rework effort of designs in manufacturing is the integration of a manufacturability analysis in the design process [GRD+97]. Manufacturability analysis enables designers to detect

production problems early in the design phase and change their design appropriately, before they finally hand it over to production planners. The analysis is based on information on the capabilities of the production unit and the constraints defined in production plans. Manufacturability analysis is in the meantime a well-known best practice in the manufacturing industry. Manufacturability is a well-recognized quality attribute of product designs.

Key Contribution

C1: Architecture-Centric Producibility Analysis

In Software Engineering, architectures are not systematically analyzed with respect to a potential misalignment with the project plan. Hence, the key solution idea in this thesis is to introduce a so-called architecture-centric producibility analysis as a mediator between architectural design and project planning. The architecture-centric producibility analysis transfers the best practice of manufacturability analysis known from the manufacturing industry to software. The producibility analysis is called architecture-centric because it uses the architecture to investigate the alignment with the current version of the project plan and tries to detect problems potentially arising during production early on. The producibility analysis detects, for instance, if the structure of the system causes problems with respect to the release plan, the resource assignments, etc. The results give architects and project planners the chance to modify the architecture and/or the project plan accordingly before any time or effort is wasted.

Producibility is the quality attribute characterizing the alignment or misalignment of architectures and project plans. It is introduced in detail in Chapter 4 when a quality model of producibility is presented.

The architecture-centric producibility analysis is based on the assumption that architectural design and project planning are and will be two separate activities in projects as architects and project planners in general have different backgrounds and concerns. Nevertheless, they need to collaborate closely in software production, which is supposed to be enabled by explicitly establishing producibility analysis as common activity.

Figure 5 illustrates the envisioned approach of an architecture-centric producibility analysis. Architectural design and project planning are typically conducted in an iterative way. Between two architectural design respectively project planning iterations, the producibility analysis can be integrated. The producibility analysis takes the current version of the architecture and evaluates the producibility the current version of the project plan. The producibility analysis identifies critical architectural and project planning elements and provides guidance for a detailed analysis of them based on checklists. The result of the producibility analysis is a list of critical elements of the architecture and the project plan potentially causing production problems and recommendations how to deal with such criti-

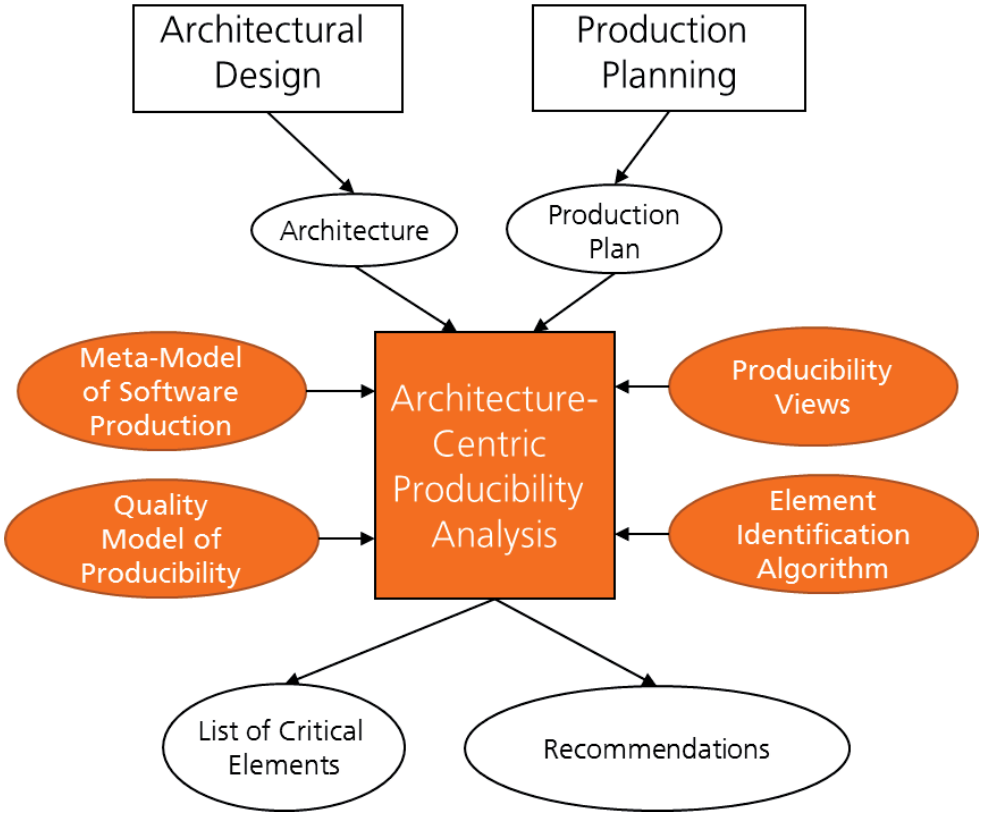


Figure 5: Overview Solution Ideas

cal elements. The recommendations are fed back to architectural design and project planning to revise the architecture or the project plan in an up-coming iteration.

Technical Contributions

The method for an architecture-centric producibility analysis is the key methodological contribution of this thesis. Several technical contributions are required to enable an architecture-centric producibility analysis.

C2: Meta-Model of Software Production

The meta-model of software production forms the basis for a better integration of architectural design and project planning and hence a producibility analysis. It defines the key concepts of architecture and project plans and their interrelationships.

C3: Quality Model of Producibility

The quality model of producibility defines producibility as a quality attribute characterizing the alignment of architectures and project plans. Based on the meta-model of software production it describes how architectures and project plans need to be aligned to increase producibility.

Hence, it forms the basis to identify production problems and provide recommendations in the producibility analysis.

C4: Producibility Views

Producibility views document the relationship of architecture and project plans. They are the basis to perform the producibility analysis and determine the producibility according to the quality model of producibility.

C5: Element Identification Algorithms

An algorithm that systematically identifies potential critical elements in the architecture and the project plan supports the producibility analysis. The algorithm captures the part of the producibility analysis that will be automated in a first step.

1.5 Benefits and Research Hypotheses

Several major benefits are expected from conducting an architecture-centric producibility analysis. In this section, the benefits are discussed and research hypotheses capturing the expected relationships of the contributions of this thesis and the practical and scientific problems are presented.

Early detection of potential production problems

One major expected benefit is the early detection of potential production problems. The architecture-centric producibility analysis can detect production problems before production has started. Already during initial architectural design activities in a project, a producibility analysis can be conducted as long as the respective information from project planning is available. Early detection enables architects and project planners to elaborate solutions before lots of time and effort are wasted in detailed design or detailed planning. Production risks can be identified and mitigated early on. Architects get the chance to compare architecture alternatives with respect to their alignment with the project plan and take architectural decisions under consideration of producibility requirements.

Check of the architecture for "completeness"

An architecture-centric producibility analysis is essentially based on the architecture and the existing documentation. While identifying potential production problems, the architecture respectively its documentation is inspected from different perspectives of roles involved in production, for instance, implementers or testers. Production problems can only be detected, if the information relevant for production is already contained in sufficient detail in the current architectural design. From the perspective of implementers or testers, this means that it is implicitly checked if all the information required to implement or test architectural elements are

contained in the architectural design. Such information includes, for instance, interfaces of architectural elements, technology decisions, constraints for detailed design of architectural elements, etc. Hence, the architecture is checked for completeness with respect to the information required by implementers or testers. This fact could also be called production readiness of the architecture.

The following hypotheses are the basis for the validation of the thesis described in Chapter 6.

H1 – Effectiveness of the Producibility Analysis Method with respect to Time and Effort: *The producibility analysis method reduces time and effort spent on production (i.e., in this case the set of all activities conducted after architectural design and project or production planning) by at least 25%.*

H1 is the basic hypothesis of this thesis with respect to the practical problem identified in Section 1.1. It is assumed that architecture-centric producibility analysis has a positive effect on the practical problem and reduces project delays and effort overhead caused during production by the misalignment of architecture and project plan. It is assumed, that architects and production planners are able to identify solutions to reduce the misalignment of architecture and project or production plan by means of the guidance provided by the producibility analysis method.

The following hypotheses H2 and H3 are more related to the scientific problems identified in Section 1.3.

H2 – Completeness of the Identification of critical Elements: *The producibility analysis method detects at least 75% of critical elements (including architectural and project or production planning elements).*

Thereby, elements are called critical if they bear the risk of causing delays, effort overhead, or quality issues during production.

H2 relates to the completeness of the producibility analysis with respect to the critical elements. The producibility analysis aims at identifying as many critical elements as possible. However, a certain number of critical elements are not expected to be detected up-front as the related problems are caused by unforeseen events occurring during production.

H3 – Correctness of the Identification of critical Elements: *At least 90% of the elements identified by the producibility analysis as critical are critical in the end, i.e. less than 10% of the identified elements are false positives and not causing any production problems.*

H3 is related to the correctness of the results of the producibility analysis. There is a certain likelihood that architectural elements are classified as

critical although they do not cause problems during production. Potential reasons for not causing production problems could be project team members performing better than expected or changes in the project context that compensate certain problems. The following section provides an overview of the research approach chosen in this thesis.

1.6 Research Approach

The research approach illustrated in Figure 6 has been applied to come up with the results of this thesis.

State of the Practice Observations

The practical problem underlying this thesis has been observed in a series of industry projects at Fraunhofer IESE, in this case architecture assessments. Since several years, we conduct architecture assessments for industrial customers. Thereby, the motivation of our customers to conduct an architecture evaluation are typically urgent problems compromising the success of running projects, for instance, doubts on the appropriateness of the architecture, quality issues regarding the final product, technical issues, or huge delays in delivery. Based on the results of our architecture assessments, management decisions concerning the continuation of the respective projects have been taken. We experienced that problems often can only be explained by combining observations made regarding the architecture with observations made regarding project plans and processes in an organization. Hence, we concluded that the architecture needs to be evaluated more thoroughly under consideration of project plans, processes, and organizational aspects.

State of the Art Literature Survey

Based on the experiences made in industry projects a literature survey has been conducted. The survey focused on research results regarding the relationship of software architecture, project plan, processes, and organizational aspects and in general on approaches aiming at aligning architecture, project plans, processes, and organization to make software development more productive. As Software Engineering adopted practices from other engineering disciplines already before, the survey also considered related literature from other engineering fields, i.e. the manufacturing industry. The survey inspired us to come up with the notion of software production as defined in this thesis and to elaborate the general solution idea to conduct a producibility analysis of software architectures.

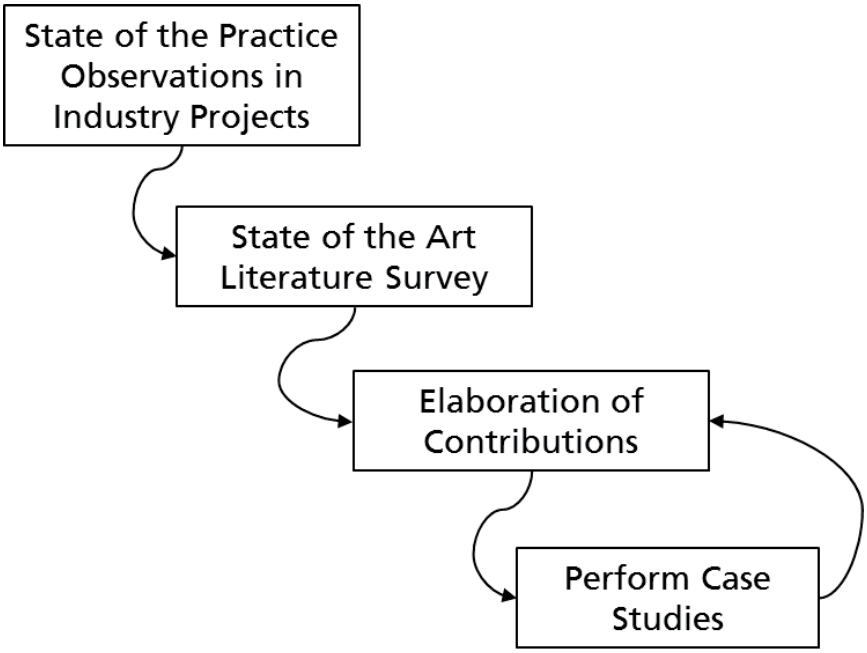


Figure 6: Overview Research Approach

Elaboration of Contributions

Based on the results of the state of the practice observations and the state of the art survey the core contributions of the thesis have been elaborated. First, the meta-model and the quality model have been derived and the idea of the producibility analysis method has been elaborated in detail. Based on the meta-model and the quality model, the producibility views and the element identification algorithm have been derived.

Perform Case Studies

According to the approach of Experimental Software Engineering [Bas93] we follow at Fraunhofer IESE, the producibility analysis method has been initially evaluated in a case study. The project to conduct the case study has been selected in this case based on its suitability but also availability when this research has been evaluated. Unfortunately, only one suitable project has been available for evaluation in this case.

The following section summarizes this introduction chapter before Section 1.8 gives an outlook on the following chapters.

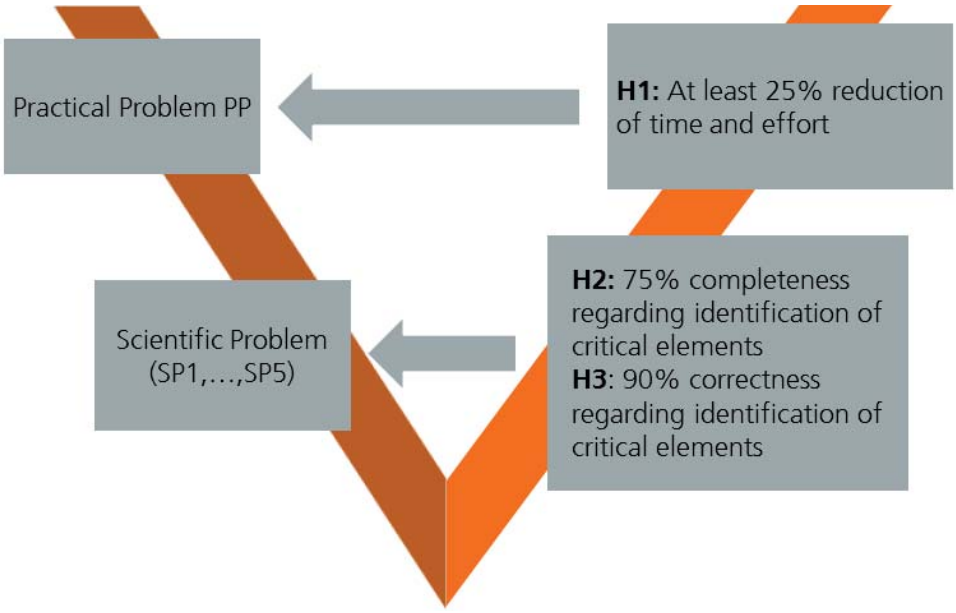


Figure 7: PhD V-Model - Relationship of Problems and Hypotheses

1.7 Summary

This chapter introduced the practical and scientific problems addressed in this thesis as well as the contributions and research hypotheses. Figure 8 provides an overview of the practical and scientific problems, the contributions and the overall relationships. The practical problem identified in various architecture assessment projects in industry is caused by five scientific problems SP1,…,SP5 elaborated based on the current state of the art reported in literature. The scientific problems are addressed by five contributions C1,…,C5. As it can be seen in Figure 8, the contributions especially focus on the scientific problem SP2 as SP2 has been figured out as the key technical problem to be addressed.

The relationship of the research hypotheses to the practical and scientific problem is shown in Figure 7. H1 relates to the practical problem of this thesis whereas H2 and H3 refer to the scientific problems identified. The especially refer to the scientific problem SP2, but implicitly also to the other scientific problems as they are all closely related.

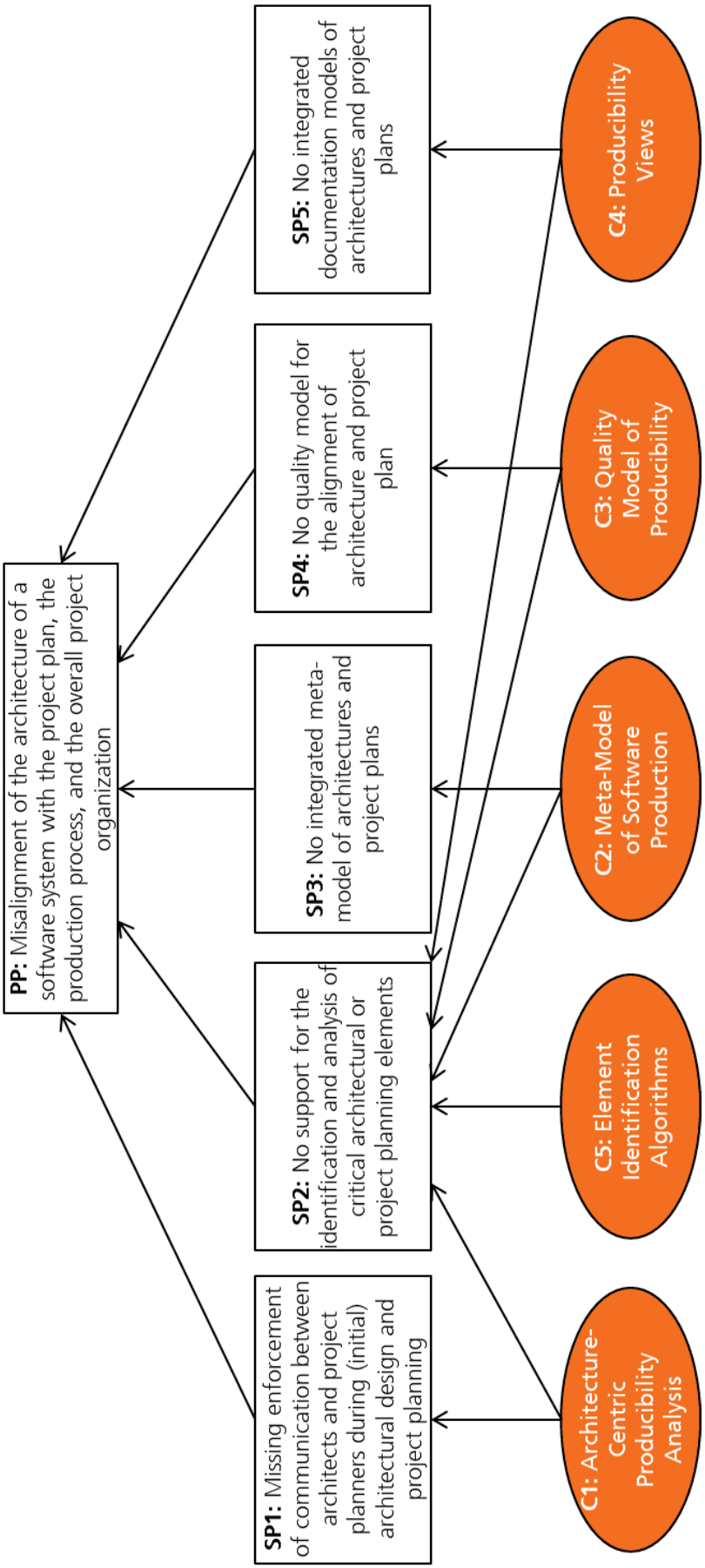


Figure 8: Overview Problems and Contributions

1.8 Outline

The remainder of this thesis is structured as follows:

Chapter 2 introduces the foundations of software architecture and software project plans including respective meta-models. Such meta-models are the basis for the meta-model of software production.

Chapter 3 defines and discusses software production. The meta-model of software production integrating the meta-models of software architecture and software project plans is introduced.

Chapter 4 defines producibility as the quality property characterizing the alignment of software architecture and software project plan. The quality model of producibility defines producibility metrics based on the quality model of software production and introduces a set of context factors influencing producibility.

Chapter 5 describes the producibility analysis method in detail. Producibility views are introduced and the algorithm identifying critical elements based on the producibility views is presented. All phases of the producibility analysis method and the provided guidance are explained. The main features of the existing tool prototype are presented.

Chapter 6 documents the validation activities that have been conducted in the course of this dissertation research and gives an outlook on future validation activities.

Chapter 7 summarizes this thesis and gives an outlook on future work.

2 Foundations and Meta-Models

This thesis addresses the alignment of software architecture and software project plans. As a basis for the introduction of the idea of software production (Chapter 3) and the quality attribute producibility (Chapter 4), this chapter presents foundations of software architecture and software project planning. Key terms and concepts of software architecture and software project plans are related to each other in meta-models of software architecture and software project plans. . These meta-models are specifically required as a basis for the meta-model of software production (Section 3.4).

2.1 Software Architecture

Software architecture plays a central role in every software project. Every software system has an architecture, no matter if it is explicitly documented and understood [RW05]. Architecture can serve various purposes if it is explicitly used in a software project as a conceptual tool. In this section, software architecture is defined and its role in software projects is discussed in more detail in Section 2.1.1. Architectural elements as the major building blocks of a software architecture respectively software system are introduced in Section 2.1.2. Section 2.1.3 generalizes architectural elements into architectural element types that are present in architectural styles, reference architectures, and product line architectures. Finally, Section 2.1.4 provides an overview on architecture documentation. The concepts introduced as part of our underlying meta-model are selected for the context of this thesis and provide exactly the concepts required for its contributions. Another meta-model developed with different quality properties in mind is, for instance, the Palladio Component Model (PCM) for component-based software architectures [RBH+07] [BKR09].

2.1.1 Definition and Role of Software Architecture

We mainly refer to the definition by Bass et al. as already mentioned in Chapter 1:

Definition Software Architecture: *"Software architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them."* [BCK03]

The tangible result of designing a software architecture is an architecture document. But software architecture is neither only an artifact derived at some point in time in a software project nor only a phase in a software project that is conducted between requirements engineering and construction. Software architecture is an ongoing activity that impacts all other activities in a software project. Typically, a whole architecture team takes care of the architecture for a specific system throughout a project. But there should always be one responsible software architect for the architecture of a certain system.

Transition from Problem to Solution Space

By defining the structure of the system and the properties of the software elements based on the requirements in a system, a software architect performs a transition from the problem to the solution space. The architect decides how solutions to customers' problems can be technically realized. But architecture still abstracts from technical details that can be decided by designers or implementers if they are not highly relevant for the overall system quality or project success [CBB+03]. Hence, software architecture addresses a higher level of abstraction than, for instance, object-oriented design.

Architecture as a Mediator

Architecture can be seen as a mediator between customers or other business-related stakeholders like project or product managers and the developers responsible for the implementation (see Figure 9).

Architecture facilitates Communication

In that sense, software architecture facilitates communication in the development team and between the various stakeholders involved in a software project that are interested in the software architecture. Thereby, "*a* stakeholder in a software architecture is a person, group, or entity, with an interest in or concerns about the realization of the architecture" [ISO07].

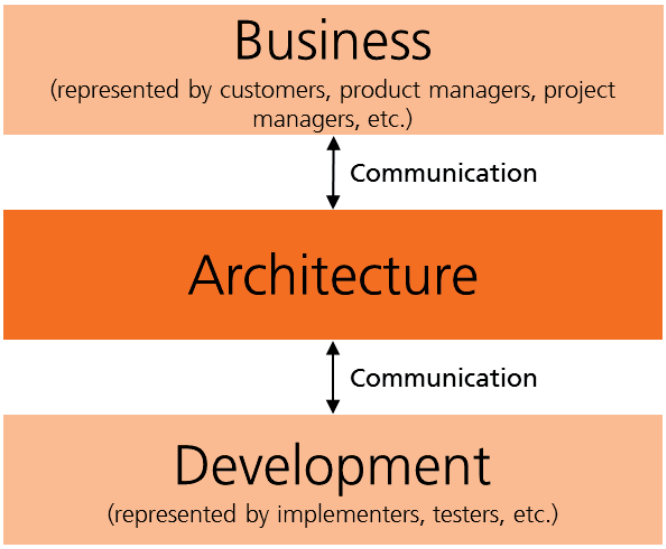


Figure 9: Architecture as a Mediator

Architects collect the concerns of the stakeholders interested in a software system. Thereby, a concern is "a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture" [ISO07].

Based on the stakeholder's concerns architects can derive an architectural design from it under consideration of certain trade-offs, where a trade-off is "a situation that involves losing one quality or aspect of something in return for gaining another quality or aspect. It implies a decision to be made with full comprehension of both the upside and downside of a particular choice" [Wiki11].

Trade-offs are required to balance eventually conflicting concerns of the stakeholders. The resulting architecture is then a means for discussion between stakeholders, but also to communicate the technical decisions to the development team. In the case of software production, architecture specifically has to enable communication between architects, project planners, suppliers and all other roles involved in production like implementers, testers, operators, etc. In that sense, it plays a vital role to discuss production requirements and their potential impact on the architecture and exchange knowledge related to production between the respective stakeholders.

Architecture enables Reasoning and Prediction

Based on the stakeholders concerns the system has to fulfill certain quality requirements related to run-time quality properties like performance, security, availability, or development-time quality properties like maintainability, flexibility, or testability [ISO01]. Unsatisfied quality requirements are a major source of project failure. The decisions taken by the architect either explicitly or implicitly affect the fulfillment of the quality requirements. Consequently, the architecture is also a means to reason about and predict the fulfillment of certain quality requirements. Note, that architecture cannot guarantee the fulfillment of quality requirements as during production many mistakes can be done by developers, but without a solid architecture, the required quality cannot be achieved in today's large and complex systems.

Architecture constrains and guides Production

The architecture constrains the production of a product. According to [JRL00], "architecture is a set of concepts and design decisions about structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant, explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains". Production in this case is referred to as concurrent engineering.

By defining the structure of the system, the architect defines which architectural elements need to be produced to come up with a running system. Hence, concrete work activities can be derived from the architecture during project planning.

Ran's definition of software architecture introduces the concept of texture. Texture defines the recurring micro-structure that is inherent to architectural elements defined by the architect [JRL00], i.e., texture refers to the internal structure of architectural elements. Although defining the internal structure of architectural elements is typically supposed to be left to the respective designer, textures are important to constrain production, in this case internal design and implementation of architectural elements. By constraining production via textures, the architect can make sure that architectural elements are realized uniformly and that certain cross-cutting features, i.e. features that typically cannot be realized within one architectural element, are realized consistently in each architectural element, for instance, logging or exception handling.

Architecture constrains and guides Project Planning

The structure defined by an architect and other architectural decisions like choosing technologies to implement certain architectural elements influence project planning. Structural decisions affect, for instance, the work-breakdown structure (see Section 2.2.2). Technological decisions influence, for instance, the selection of developers, i.e., resource assignment (see Section 2.2.6). Each element foreseen in the architecture needs to be assigned to a developer with appropriate skills regarding the chosen technologies.

The following section introduces architectural elements, i.e. the major buildings blocks of architectures respectively software systems.

2.1.2 Architectural Elements

We call the software elements mentioned by Bass et al. that make up a software architecture architectural elements in this thesis. Thereby, architectural elements are defined as follows:

Definition Architectural Element: *"An architectural element is a fundamental piece from which a system can be considered to be constructed."* [RW05]

Architectural elements can be, for instance, modules, components, connectors, or deployment units.

Architectural elements can be recursively refined to enable a hierarchical decomposition of a system to be able to deal with the overall complexity. Furthermore, they can be related in various ways that will be discussed later in more detail.

All architectural elements have various properties. Properties of architectural elements are defined as follows:

Definition Architectural Element Property: *Architectural element properties are characteristics of architectural elements that need to be considered while dealing with them in a software project.*

Examples for architectural element properties can be the technologies selected by the architect to realize the architectural element, the estimated size or complexity, etc.

Most of the time, we can abstract from the differences between certain architectural elements in this thesis, for instance, between modules and components. Nevertheless, definitions of module, component, connector, and deployment unit are provided in the following to point out the differences and provide examples for architectural elements.

For a definition of modules, we refer to the definition given [CBB+03]:

Definition Module: *"A module is an implementation unit of software that provides a coherent unit of functionality"*

Each module has an interface that specifies its responsibilities. Modules are the units assigned to developers for implementation. Production properties of modules can be, for instance, a selected programming language, coupling with other modules, cohesion, but also their potential to reuse parts of other modules for their realization. Such properties can influence, for instance, the selection of developers responsible for the module or the order of realizing certain modules.

Components are often called run-time entities as they make up the executable software system by interacting in a predefined way via clearly defined interfaces. Various definitions of the term component have been given. One that we specifically want to mention here as it states several properties of a component that are relevant for production is given by Szyperski [Szy02]:

Definition Component: *"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Contractually specified interfaces of components are an important property as contracts can be leveraged to specify a component's behavior quite formal and involve, for instance, third parties in the production process easier. Hence, organizations are enabled to set up supply chains of components, which is an important factor also for production in other engineering disciplines like manufacturing. Besides contractually specifying the interfaces additional production properties of components could be specified like, for instance, maximal lead time, i.e. the time it may take to deliver a component, etc.

Components are in general made up of modules, but the modules and the way the component has been built out of them is no longer visible as they are typically delivered as binaries that come out of a compile respectively build process.

Connectors are interaction mechanisms for components. They realize the communication between components according to certain protocols and make sure that data is transferred between them appropriately. In [TMD10], connectors are defined as follows:

Definition Connector: *"A software connector is an architectural element tasked with effecting and regulating interactions among components."*

Hence, connectors are first class entities at runtime together with the components. They are extremely important to assure run-time properties of the system like performance, reliability, availability, or security and can get quite complex. Connectors should specifically be considered during software production because of their importance regarding the quality of the software product, but also because of their potential complexity. Their complexity is a potential source of production problems. Hence, architectural decisions regarding connectors, for instance, to build connectors on top of certain middleware technologies should be well considered.

Deployment Units are defined as follows in this thesis:

Definition Deployment Unit: *Deployment units are the architectural elements of a software system packaged to be shipped and installed.*

Hence, deployment units are the architectural elements that are handed over to customers or operators of a system. They are mainly made up of components and connectors as run-time entities, but can also contain, for instance, configuration files in addition. Deployment units are installed in the selected runtime environment on hardware components hosting such runtime environments. The ability to create deployment units at fixed points in time can be critical during software production as sometimes operators can only install deployment units at specific points in time during operation.

Layers and Clusters

Besides such atomic architectural elements like modules, components, connectors, and deployment units, higher level architectural elements like layers, clusters, or subsystems exist to structure today's large and complex systems. A layer, for instance, is "a collection of code that forms a virtual machine and that interacts with other layers only according got predefined rules" [CBB+03]. Layers, clusters, or subsystems can be interpreted as aggregations of finer grained architectural elements. Layers are used to define horizontal structures in an architecture. Upper layers are allowed to access lower layers but not vice versa. Layers ab-

stract from technical details of underlying architectural elements. Clusters are a way to vertically structure an architecture. They can be used, for instance, to encapsulate cohesive sets of functionality that each can span all layers of a system or architecture. For a detailed discussion of layers and clusters (also called slices), we refer to [CBB+03].

Figure 10 visualizes the concepts of the overall architecture meta-model that have been introduced so far.

Architectural Element Relationships

As mentioned above, architectural elements can be related to each other in various ways. Besides the part of relationship and a general relationship that have been defined above on architectural elements, the specific architectural elements can have specific relationships as shown Figure 11. A specific relationship defined on modules is inheritance. Furthermore, modules can use each other which enables reuse. Components can call each other to realize the overall behavior expected from the system. Per definition, connectors are related to components. Each connector relates at least to two components. Components and connectors are realized by modules. Components are manifested in deployment units. Note that potentially one component can be part of several deployment units of a software system.

Architectural element relationships are of particular interest for software production. It has already been mentioned before that modules are units of work which can be assigned to developers during production. Consequently, the relationships between modules, for instance, impact the communication paths in the team and thus the production process. The

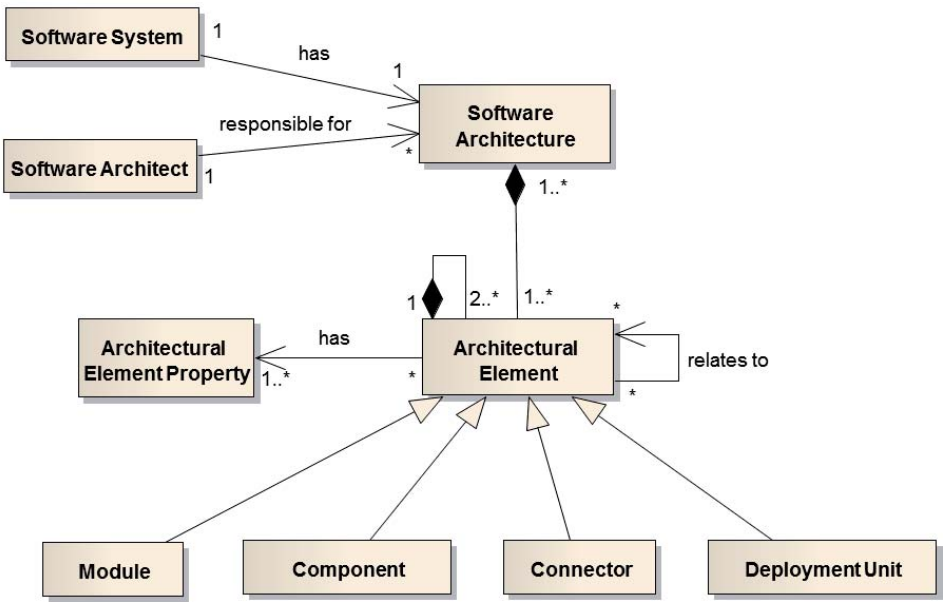


Figure 10: Core of Architecture Meta-Model

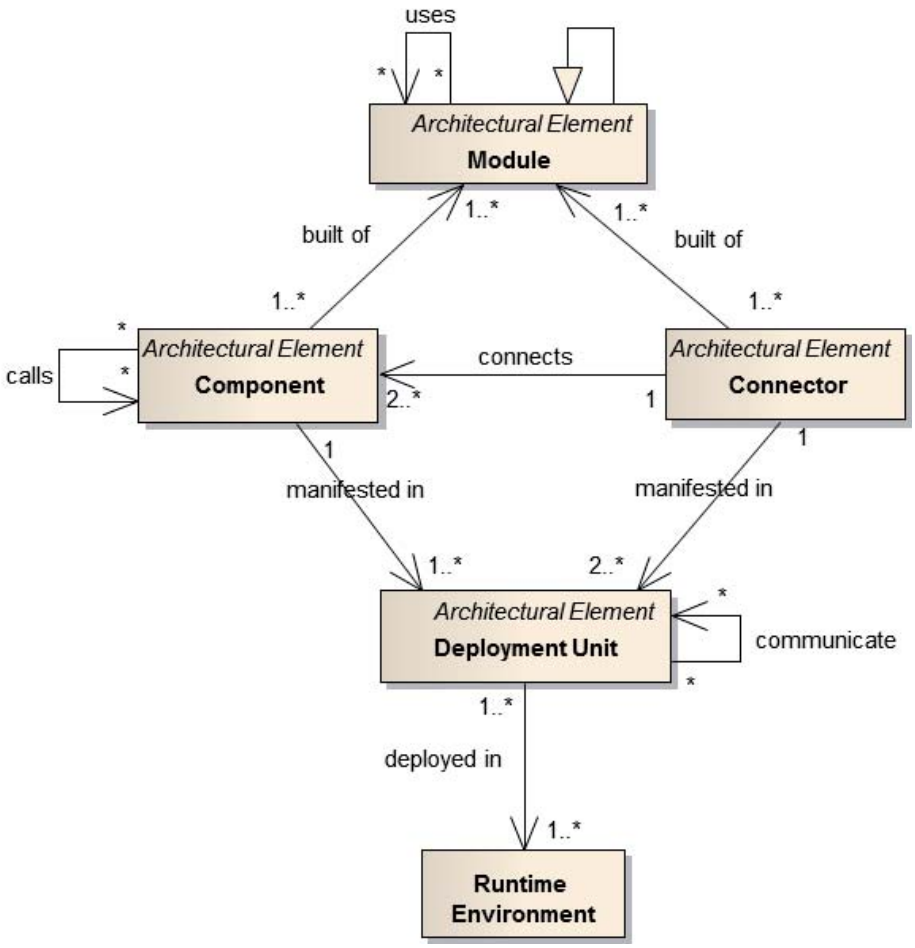


Figure 11: Architectural Element Relationships

relationships between modules and components, for instance, impact building and testing. If many modules are required to build a certain component the build process itself can get complex. This is also a potential source of delays during production because many modules have to be realized before the component can be built and tested. As soon as one module is not finished yet the upcoming production work activity potentially has to wait. Components can be part of many deployment units as mentioned above. Such components are candidates for early completion and thorough quality assurance as they can cause damage at different places in the software system.

In the following section, architectural element types are introduced as a generalization of architectural elements present in architectural styles, reference architectures, and product line architectures.

2.1.3 Architectural Element Types

This section introduces architectural element types. Architectural element types are defined as follows:

Definition Architectural Element Type: *An architectural element type is general type of architectural element recurring in single system architectures following a certain architectural style as well as in architectures conforming to the same reference or product line architecture*

Before we discuss architectural element types in more detail, we define the architectural style, single system architecture, reference architecture, and product line architecture as referred to in the definition of architectural element type.

Architectural Style

An architectural style (also called architectural pattern) according to [BCK03] is defined as follows:

Definition Architectural Style: *An architectural style "is defined by:*

- *a set of element types (…)*
- *a topological layout of the elements indicating their interrelationships*
- *a set of semantic constraints (…)*
- *a set of interaction mechanisms (…) that determine how the elements coordinate through the allowed topology."*

Prominent examples for architectural styles are client server style, pipe-and-filter style, or peer-to-peer style.

The definition of architectural style is shortly illustrated using the pipe-and-filter style. The (architectural) element types defined by the pipe-and-filter style are pipes and filters. More precisely speaking, pipes are connectors and filters are components connected via pipes. The topological layout mentioned in the definition is shown in Figure 12.

Filters receive data via pipes, transform such data in some form, and send the data to the next pipe. Filters are stateless and not aware of the next filters that will process their output, which is a semantic constraint

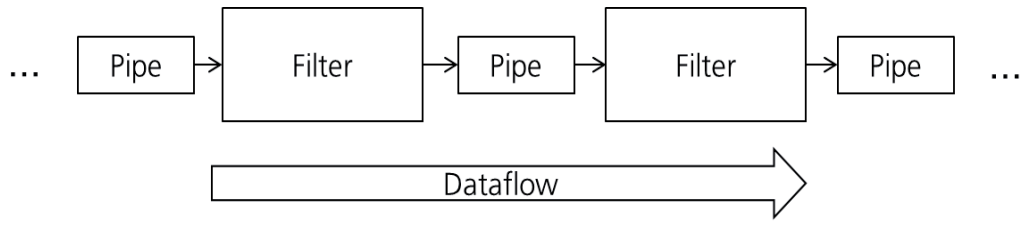


Figure 12: Pipes and Filters

in this case. An interaction mechanism has to be selected in a concrete case to specify how pipes and filters exchange data.

Architectural styles capture architectural knowledge that can be reused by architects across various systems. Architectural styles have been published in several architecture handbooks, for instance, in [BMR+96].

Single System Architecture

A single system architecture is defined as follows in this thesis:

Definition Single System Architecture: *A single system (or product) architecture (SSA) is a software architecture designed based on the requirements in one single software system or product.*

A SSA is a special purpose architecture in a sense that there is no guarantee that it can be reused for another software system without further ado. SSAs can be designed using common architectural styles, patterns, and tactics, but their combination is somehow unique. The definition of software architecture of Bass et. al. [BCK03] cited before first of all refers to SSAs.

Reference Architecture

Reference architectures are blueprints or templates for software systems of a specific domain. In [HND99], the following definition of reference architecture is given:

Definition Reference Architecture: "*A reference architecture defines element types, allowed interactions, and how the domain functionality is mapped to architectural elements*."

A reference architecture is designed based on typical domain requirements and domain knowledge and sketch architectures of products in a specific domain on a higher level of abstraction. Hence, they are often also called domain specific architectures [JRL00]. Reference architectures can have different shapes. They can be described, for instance, as an architectural style or sketch conceptual architectural elements for systems in a specific domain. Examples are the Open Group Service-oriented Reference Architecture [OpenGr09] or the Quasar reference architecture [Sie04].

SSAs can be based on reference architectures, but refinements are typically necessary in the concrete case to derive a SSA from a reference architecture that fulfills the concrete requirements in a product of the respective domain.

Product Line Architecture

A software product line architecture (PLA) is an architecture for a product line of software systems. Thereby, a software product line is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [CN02].

Definition Product Line Architecture: *A product line architecture is a "core asset that is the software architecture for all the products in a software product line. A product line architecture explicitly provides variation mechanisms that support the diversity among the products in the software product line"*. [NC07]

All members of a software product line share the same PLA. This is achieved by explicit variation points in the PLA, which allow architects to instantiate the PLA for specific members of a product line.

A PLA is different from a reference architecture in a sense that it is built based on a concrete set of requirements in a concrete set of products whereas a reference architecture is designed based on rather fuzzy domain requirements. The difference of a PLA to a SSA is the variability that is contained in the architecture and that needs to be resolved in the concrete case. In [Perry98], different ways of capturing variability in PLAs are described. One way of capturing variability that is mentioned there is to use an architectural style defining architectural element types, constraints, etc.

Architectural Element Types

The definitions of architectural style, reference architecture, and product line architecture all refer to architectural element types and vice versa. Architectural element types play an important role in software production, as we will see later in Chapter 3. Production processes contain procedures called production work activities that guide developers in producing certain architectural element types.

Figure 13 shows an excerpt of the architecture meta-model including architectural element types and their relationship to architectural elements, architectural styles, reference architectures, and product line architectures.

The following section gives an overview on architecture documentation.

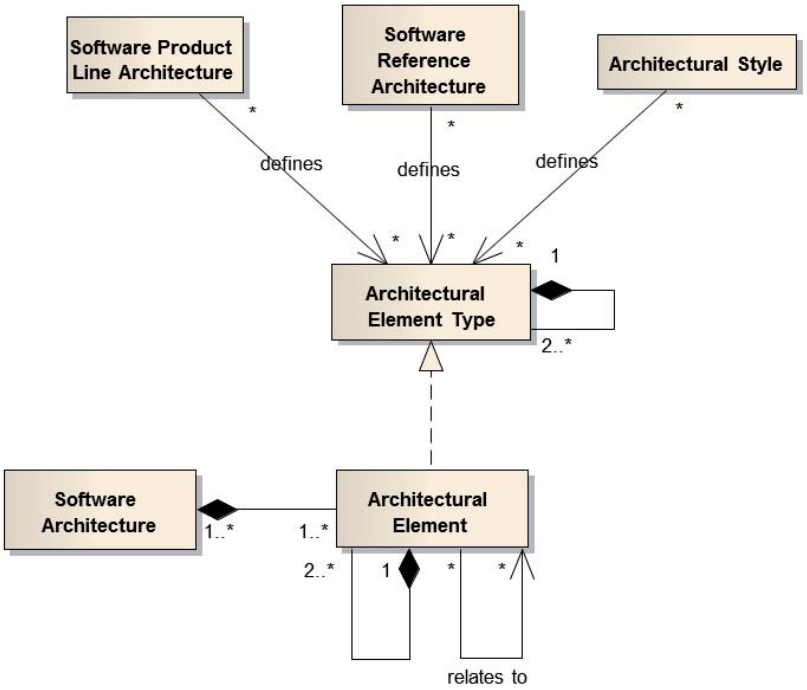


Figure 13: Architectural Element Types

2.1.4 Architecture Documentation

A recommended best practice to document architectures is using architectural views [CBB+03]. Architectural views are defined as follows:

Definition Architectural View: *"An architectural view is a representation of a set of system elements and the relationships associated with them."* [CBB+03]

The rationale for using views is the inherent complexity of software systems and their architectures. Views handle the complexity by focusing on a specific perspective on the system and the respective elements in each view. Each view is typically relevant for a subset of the overall set of stakeholders.

Several view models have been published, one prominent one being Kruchten's 4+1 viewmodel [Kru95]. In [CBB+03], three types of views are introduced, namely module, component and connector, and allocation. We refer to this classification in this thesis and define the respective views as follows:

Definition Module View: *A module view "enumerates the principal implementation units, or modules, of a system, together with the relationships among these units"* [CBR+03].

A module view show the module structure of a system, which could be a view showing inheritance relations as well as a view showing the functional decomposition into modules.

Definition Component and Connector View: "*A Component and connector view provides a picture of runtime entities and potential interaction.*" [CBR+03]

Definition Allocation View: An allocation view *"presents a mapping from the elements of either a module or a component and connector style onto the elements of the environment"*. [CBR+03]

One example for an allocation view is a deployment view showing the mapping or allocation of deployment units to hardware components. Another example is a resource allocation view visualizing how resources of the development organization are assigned to modules.

The documentation of views is one important part of architecture documentation. Furthermore, architecture documentation should contain additional documentation including, for instance, architectural scenarios and architectural decisions.

Definition Architectural Scenario: *An architectural scenario is a precise description of an anticipated situation of usage, operation, or development that a system respectively its architecture is likely to face, along with a precise description of the desired response to the situation.*

Architectural scenarios are a means to formulate the requirements in the architecture. Besides architectural design, architectural scenarios are used by many architecture analysis methods like the Software Engineering Institute's Architecture Trade-Off Analysis Method (ATAM) [KKC00] or the Software Architecture Analysis Method (SAAM) [KAB+96].

Definition Architectural Decision: *An architectural decision is a decision on the structure of a software system or properties of specific architectural elements.*

By documenting architectural decisions, rationales are given making the architecture more understandable.

Figure 14 shows an excerpt of the architecture meta-model including the concepts related to architecture documentation.

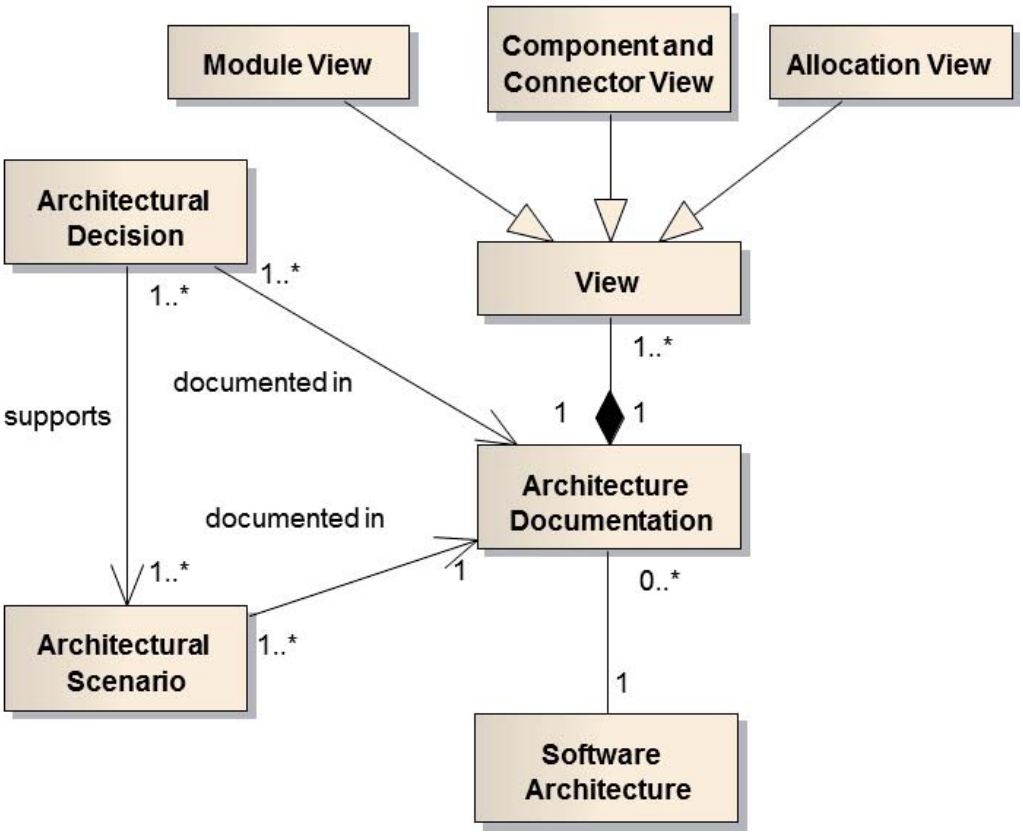


Figure 14: Architecture Documentation Meta-Model

2.2 Software Project Plans

Besides software architecture, software project plans are key artifacts in software production, as we will see later. This section introduces the foundations of software project plans and incrementally introduces the underlying meta-model used in this thesis. After defining project plans and discussing their role in software projects in Section 2.2.1, the major parts of software project plans are introduced, namely work breakdown structures (Section 2.2.2), project schedules (Section 2.2.3), development processes (Section 2.2.4), and resource plans (Section 2.2.6). Finally, we refer to how software project plans are typically documented in Section 2.2.7.

2.2.1 Definition and Role of Software Project Plans

Software project planning is the activity to come up with and maintain a software project management plan (project plan) to coordinate all the

activities in a project to achieve the project goals. As already mentioned in Chapter 1, we refer to the following definition of project plans.

Definition Software Project Plan: *"A Software Project (Management) Plan is the controlling document for managing a software project; it defines the technical and managerial processes necessary to develop software work products that satisfy the product requirements."* [IEEE98]

Based on the product requirements and the overall project objectives, project planners have to decide on project scope, project schedule, project organization and resource allocation, and development processes.

Planning reduces risk

Planning is one of the key activities in every software project. Today's software projects are often large and complex, span long periods of time, and involve many project team members from different organizational units and even organizations. Hence, project planning is critical for project success as it can significantly reduce the risks in a project. According to [Sta09], insufficient project planning is one of the major reasons for delays, budget overruns, or failure in software projects. Project planning is an ongoing activity throughout a project as project plans are typically a matter of constant change.

Planning facilitates communication

Project planning requires communication and collaboration between many project stakeholders. A project planner needs to elicit the concerns of various project stakeholders and try to fulfill them by making certain trade-off decisions. Thereby, a lot of communication is required which is beneficial for the overall project success. Stakeholders get to know each other, their concerns, and establish communication paths between them that can be used later on during project execution.

Planning constrains the work in a project

Planning constrains the work of the project team members. First of all, work activities are defined and assigned to project members. But furthermore, work activities are constrained by effort, cost, and time constraints and technical processes are prescribed that must be followed during work. Individual project team members can plan their personal work, but only within the constraints defined in the project plan.

Reuse of Experience

Project managers should make use of existing knowledge and experience from previous projects when planning a project. The Experience Factory Approach [BCR94] supports organizations in setting up an appropriate environment. Knowledge and experience is packaged and stored in an experience base. Based on the characteristics of an up-coming project various artifacts like project plans, process descriptions, architectures, etc. can be selected and retrieved from the experience base to be adapted to the project context and reused.

Figure 15 shows the general parts of a software project plan: work breakdown structure, project schedule, resource plan, and development

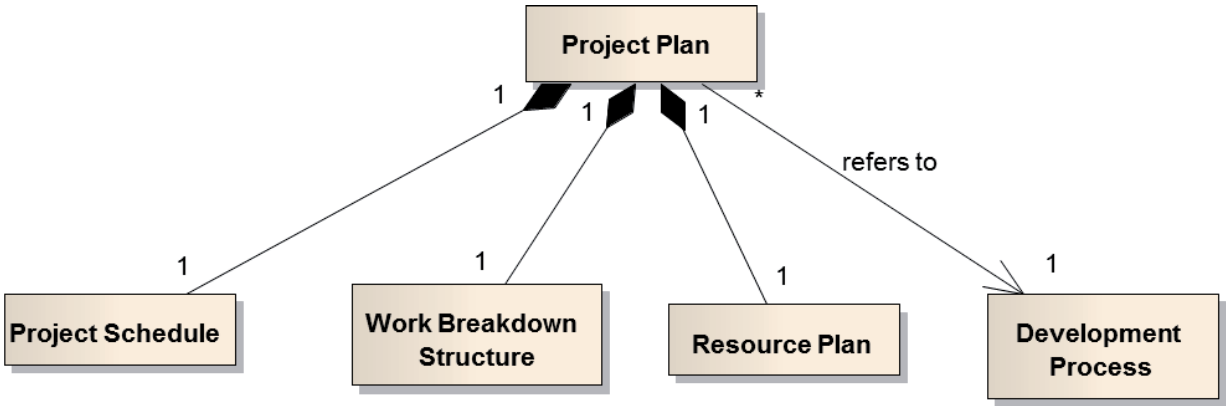


Figure 15: Project Plan Meta-Model Overview

process. In the following sections, each of such parts of a software project plan will be introduced in more detail.

2.2.2 Work Breakdown Structure

Work breakdown structures are related to the scope of a project. The project scope defines the work to be performed in a project: "The project scope is the work that must be performed to deliver a product, service, or result with the specified features and functions." [PMBOK04]

The huge amount of work that needs to be done in a software project needs to be structured. The overall work is typically decomposed into smaller units of work that are less complex and can be accomplished by certain resources. The result of such a decomposition of work is called work breakdown structure (WBS). We refer to the following definition of work breakdown structure:

Definition Work Breakdown Structure: *"A work breakdown structure is a deliverable-oriented hierarchical decomposition of the work to be executed by the project team to accomplish the project objectives and create the required deliverables. It organizes and defines the overall scope of the project…" [PMBOK04]*

A work breakdown structure consists of work activities that are defined as follows:

Definition Work Activity: *A work activity is a fixed unit of work with defined inputs and outputs.*

Definition Work Activity Property: *A work activity property is a characteristic of a work activity that needs to be considered while performing the respective work.*

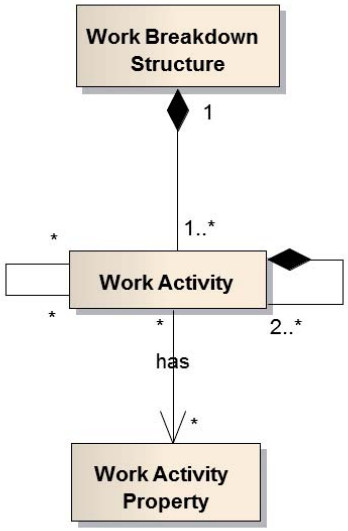


Figure 16: Meta-Model Work Breakdown Structure

Figure 16 shows how work breakdown structure, work activities, and work activity properties are related in the project plan meta-model.

Phase-orientation vs. Product-orientation

A WBS can be derived based on various strategies, two prominent ones being phase-oriented and product-oriented. If the work is organized along phases, the phases of the underlying software development process can be used for structuring on top-level, for instance, requirements engineering, design, implementation, and testing. Using a product-oriented strategy would mean to use the product to be built to organize the work in the WBS. This could either mean to use the requirements in the product for structuring and defining chunks of requirements to be realized as work units. Alternatively, the architecture of the system can be used to define work units [Fair09], i.e., architectural elements define the work units in this case. Figure 17 shows simple examples of WBS.

The selection of an appropriate decomposition strategy depends on the project context. If the waterfall model is used, organizing the project in phases according to the waterfall model on top level seems appropriate. If an iterative and incremental process model is used, the work should rather be organized product-oriented, i.e. based on product requirements or architectural elements that are realized in a certain iteration or increment. Combinations of phase- and product-oriented strategies are also feasible. After decomposing the work into phases according to the waterfall model, for instance, the implementation can be decomposed into smaller units of work along the architecture.

As we will see later, software production is always based on a product-oriented WBS. Hence, in the following we will only consider product-oriented decompositions of the work to be performed in a project.

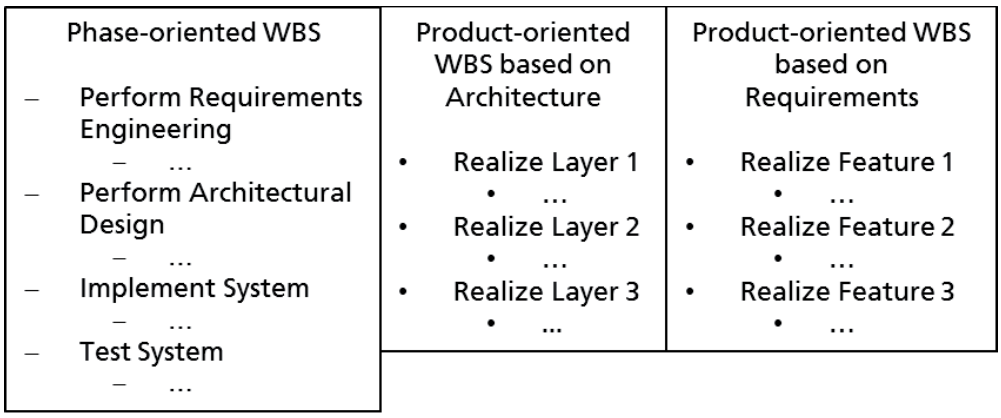


Figure 17: Example Work Breakdown Structures

The WBS introduced in this sub-section is the basis for all further project planning activities like project scheduling and resource. The following section introduces project schedules.

2.2.3 Project Schedule

A project is schedule is derived based on the WBS defined during project scoping. A project schedule is defined as follows:

Definition Project Schedule: *A project schedule defines the order of performing the work activities specified in the WBS and assigns effort and time to the work activities.*

As many projects today follow an incremental and iterative approach (and also software production is incremental and iterative as it will be introduced in Chapter 3), project scheduling includes the definition of iterations and assigning work activities to them. Iterations are defined as follows:

Definition Iteration: *An iteration is a fixed period of time wherein certain work activities are performed.*

At the end of certain iterations, releases of a product can be delivered.

Iteration and Release Planning

During iteration or release planning work activities are assigned to iterations or releases. As iterations are performed sequentially, assigning work activities to iterations leads to a certain ordering of work activities and consequently a first version of the project schedule. According to [RM05], release planning is defined as the activity of assigning requirements to releases. Assigning requirements to releases means the architectural elements required to fulfill the respective requirements are supposed to be produced in the respective release.

Iterations or releases can have fixed or variable durations. In agile software development, time-boxing is often used and all iterations in a project have a fixed duration, for instance, 2 weeks. Consequently, work units at least need to be decomposed until they fit into iterations of the respective duration.

Iterations or releases are typically planned based on the concerns of various stakeholders, the most important one typically being the customer. Especially in agile software development [Hun06], iteration or release planning heavily involves customers. But also a more technical perspective should be considered by, for instance, involving the architect into release planning as architectural decisions on the structure of the system influence release planning, as we have seen in the example in Section 1.2.

Various approaches to release planning have been proposed [SGF+10]. They can be distinguished by the factors they use to identify the requirements for certain releases. Beyond such factors are, for instance, technical, budget and cost, resource, and time constraints. Thereby, technical constraints mainly refer to requirements dependencies and potential technical problems in realizing certain requirements. Technical problems can be analyzed in detail based on the planned architecture, as the architecture specifies the technical solution. Unfortunately, the existing release planning approaches do not clarify the influence of the architecture on release planning in detail, i.e. which characteristics of an architecture influence release planning in which way. They especially provide no guidelines for architects how they should align their designs with a release plan.

Besides iterations, project schedules consist of milestones. Milestones are defined as follows:

Definition Milestone: "*A milestone is a scheduled event to measure progress.*" [IEEE98]

During project scheduling, milestones are defined. Examples of milestones can be the completion of iterations or releases, but also, for instance, the completion of certain subsystems.

Figure 18 shows the excerpt of the project plan meta-model including project schedule and the relationship to the work breakdown structure and work activities.

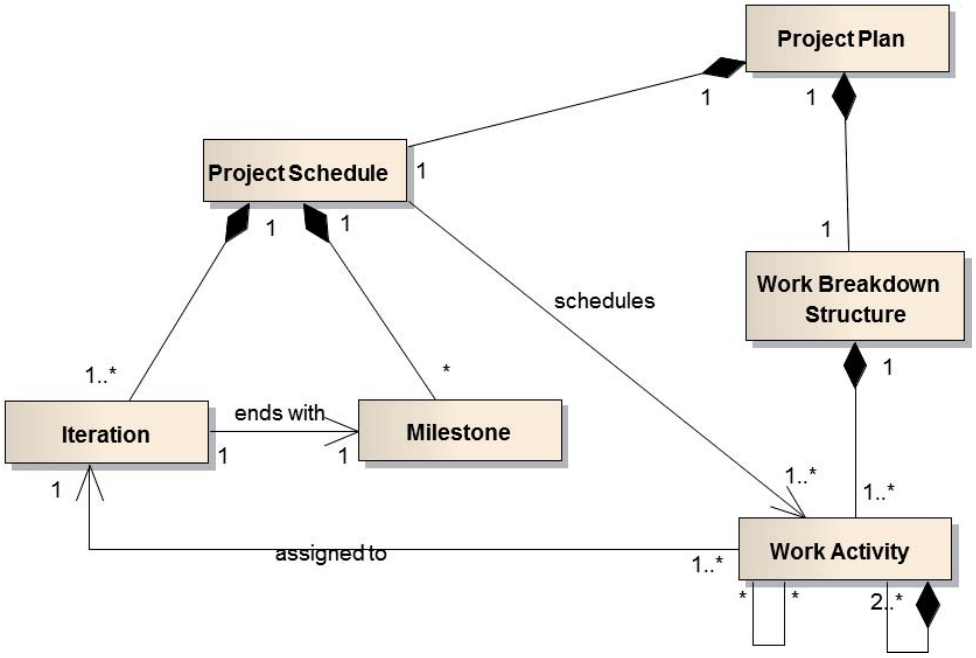


Figure 18: Meta-Model WBS and Project Schedule

Besides ordering the work units by assigning them to iterations or releases, effort, cost, and time have to be assigned to them. Hence, typical properties of work activities are effort, cost, and time. The project planner has to come up with estimates for effort, cost, and time of certain work units.

Estimation approaches

Various approaches exist to estimate effort, cost, and time to be allocated to the work units. As mentioned in Chapter 1, prominent approaches are CoCoMo (II) [BAB+00] or Function Point Analysis [ISO09]. Another prominent approach that is based on expert opinion is the Delphi Method [LT75]. According to the Delphi Method, several experts are asked to estimate the effort of the work units in the WBS. The estimates are used to come up with an average effort estimation to be used in the project schedule.

2.2.4 Development Process

A project planner or manager has to define respectively to select a development process to be used in the project. The development process defines how the work in a project has to be performed. It defines, for instance, how requirements are elicited, how an architecture is designed, and how implementation, test, and deployment of a system have to be performed in general.

We define a development process as follows:

Definition Development Process: *A development process is a set of development activity types, work products, roles, and tools that can be applied to develop a software system.*

The definition of development process refers to development activity types and work product types that will be defined in the following:

Definition Development Activity Type: *A development activity type is a procedure to perform a certain development activity aiming at creating a certain work product type.*

Examples for development activity types are use case analysis during requirements engineering, writing test cases to prepare testing, etc.

Definition Work Product Type: *A work product type is a type of artifact created by certain development activity types during a project.*

Examples for work product types are use case descriptions, test case specifications, etc.

Figure 19 shows development processes and related concepts in the meta-model underlying this thesis.

Process frameworks and best practices

The project manager can make use of various development process frameworks and select development activity types to define the development process. Popular process frameworks are, for instance, the Rational Unified Process [Kru03] or its open source version OpenUP [OpenUp], the V-Model XT [VModellXT], or agile processes like Scrum [SB01] or Extreme Programming [BA04]. Such process frameworks are built around development activity types for all phases of software development like, for instance, requirements engineering, object-oriented design, or testing.

Tools

Defining the development process for a project includes the selection of tools to be used during development. If, for instance, model-driven development is supposed to be used in a project, an appropriate tool environment including model editors, generators, etc., needs to be selected.

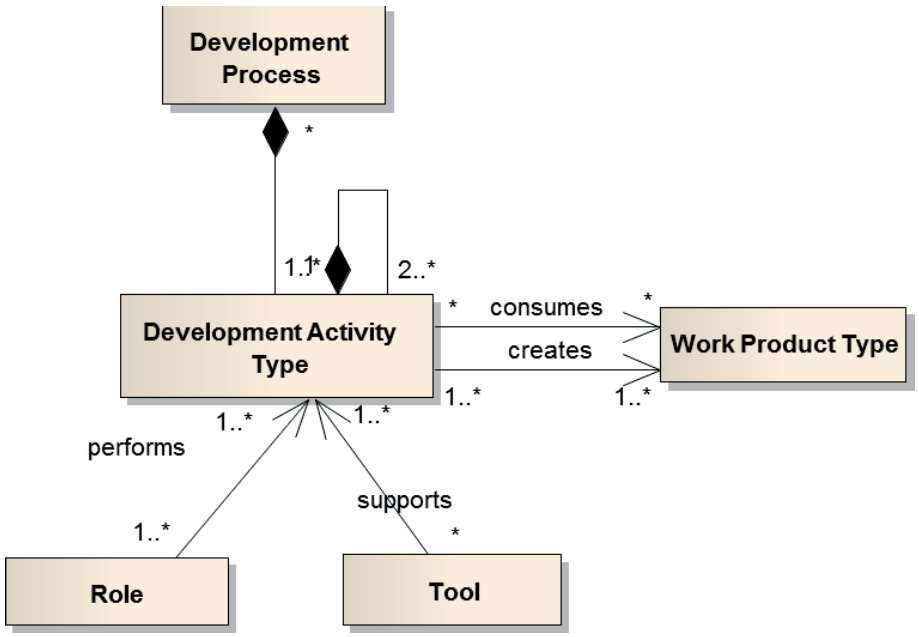


Figure 19: Meta-Model Development Process

2.2.5 Software Project Organization

Project organization is concerned with the internal structure of projects. Each project consists of a project team. We define a project team as follows:

Definition Project Team: *A project team is the collection of all human resources actively involved in work activities in a project.*

Project teams can be made up of project sub-teams.

Hierarchical Organization

Most software project teams are organized in a hierarchy. The team members on higher levels of the hierarchy are mainly concerned with management activities as they have to manage the team members on the lower levels in the hierarchy. Most of the technical activities are performed on leaf nodes.

We want to distinguish two general strategies for organizing projects in a hierarchy: role-orientation and product orientation.

Role-orientation

In the case of a role-oriented organization, the project team is organized according to the roles being part of the development process. Team members are assigned to roles, for instance, requirements engineer, designer, implementer, or tester. Then, all team members assigned to the same role form a sub-team team in the project. Hence, the overall project team is divided into a requirements engineering sub-team, an architectural design sub-team, an implementation sub-team, and a testing sub-team with respective managers organizing the work in the sub-team.

Product-orientation

Alternatively, the project organization can be product-oriented and follow the structure of the overall product to be built. If the product would be structured into three functional clusters A, B, and C, for instance, the project team could be subdivided into three teams, each of them responsible to realize one functional cluster. In [Lar10], such teams are also called component teams, as they are assigned to develop and maintain a certain component or subsystem. Alternatively, feature teams can be built that are responsible to realize certain features which can involve various components.

Project teams have certain properties. Thereby, project team properties are defined as follows:

Definition Project Team Property: *A project team property is a characteristic of a project team that needs to be considered when assigning work activities to the project team.*

Various project team properties are relevant in software projects, beyond them the internal organization of the project team, the geographic distribution of the project team, and the classification as internal or external team or sub-team. These properties will be shortly discussed in the following.

Project Team Organization

Various models for organizing decision making and work distribution inside project teams have been published, for instance, the chief programmer model [Bak72]. In the case of the chief programmer model, one experienced architect or programmer takes all design decisions and implements the key parts of a system or sub-system. The chief programmer can have one assistant replacing him if required and a librarian taking care of documentation and administrative tasks. Additional team members can be assigned to "easy" programming tasks. Another model that is more based on collaboration and consensus regarding decision making is the structured open team [Con93]. In such a team, there is a technical leader representing the team to the outside, but decisions are taken together by all team members in a democratic way. The tasks are distributed among team members by majority vote.

Geographic distribution

Today, project planners or managers should always consider the geographic distribution of a project team. Geographic distribution means that certain sub-teams or single human resources can be distributed worldwide. Such distribution brings several factors like different time zones, different cultures, etc. into play that influence the overall project.

Teams as well as human resources can be classified as internal or external. Internal teams or human resources are full members of the overall organization responsible for the project. Teams of externals can be included in a project team to take over certain work activities, but they are not employed by the organization responsible for the project.

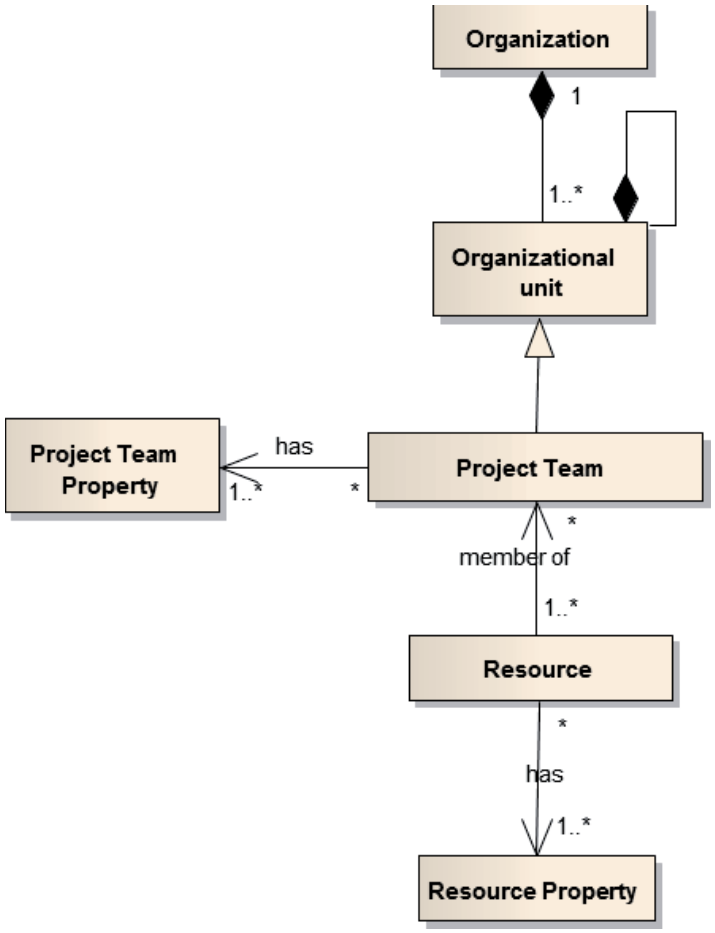


Figure 20: Organization Meta-Model

Figure 20 shows the excerpt of our meta-model dealing with project teams and related concepts.

2.2.6 Resource Plan

The resource plan is the part of the overall project plan dealing with the allocation or assignment of resources to work activities.

In the following, we define the concepts resource plan and resource assignment:

Definition Resource Plan: *A resource plan is a plan made up of resource assignments.*

Definition Resource Assignment: *A resource assignment is a decision on assigning a team or single human resources to work activities of a work breakdown structure.*

Each work activity needs an assigned resource to make sure that someone takes responsibility for a work activity and the respective work is

performed. Several resources or whole teams can be assigned to a work activity to perform the work in collaboration. But it depends on the type of work if this is reasonable. If the work cannot be split appropriately, only one resource can be assigned to the work activity.

Figure 21 shows another excerpt of the project plan meta-model underlying this thesis. The resource plan respectively resource assignment connect organizational elements to work activities.

According to [AJM06], project managers base their decisions regarding resource allocation mainly on their personal experience and subjective perception. This can be a source of productivity problems if not the best-suited resources are selected based on objective measures and if the personality of the resources is not considered.

Several objective factors influencing the allocation of resources to work activities should be considered, beyond them skills and availability of resources as well as, for instance, their geographical location. All of these factors are critical to the successful completion of certain work activities. If an assigned resource does not bring in the required skills, the quality of the result may be reduced or more time and effort may be consumed. Availability of resources is crucial for the timely completion of activities. If related activities are performed by members of different organizational units, potentially at different geographical locations, communication problems can appear and delay the project.

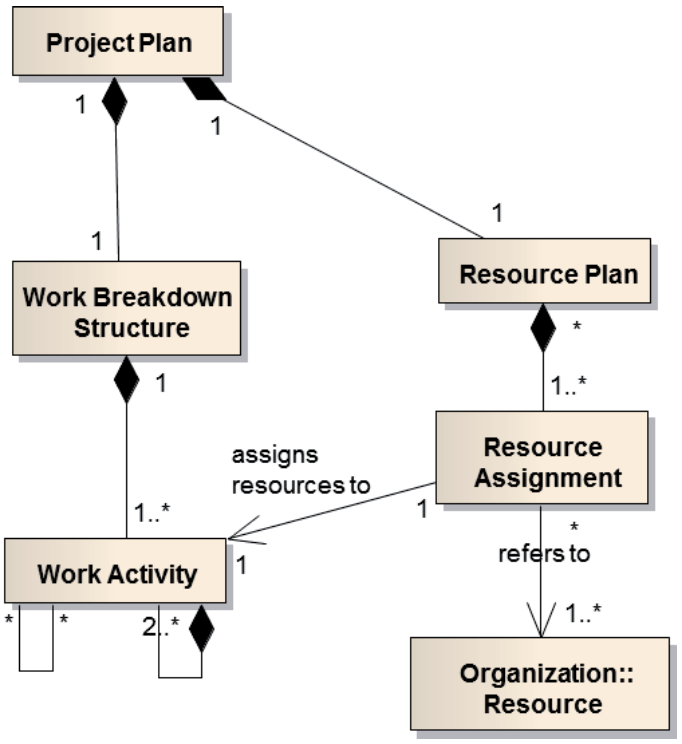


Figure 21: Resource-Plan Meta Model

The following section provides an overview on project plan documentation.

2.2.7 Project Plan Documentation

Project plans must be documented to be able to serve as a means for communication in a project. A structure for documenting a software project plan is included, for instance, in [IEEE98]. Similar to architecture descriptions, project plan documentations consist of various diagrams and textual descriptions.

WBS as a hierarchical decomposition of the overall work to be performed are often represented in a tree-structure.

Gantt Charts

One of the most prominent representations of project schedules are Gantt Charts. They are used in various project planning tools like Microsoft Project [MSProj10]. Gantt-charts are defined as follows:

Definition Gantt-Chart: *"Gantt charts are a graphic display of schedule-related information. In the typical Gantt chart, schedule activities or work breakdown structure components are listed down the left side of the chart, dates are shown across the top, and activity durations are shown as date-placed horizontal bars." [PMBOK04]*

The resource allocation defined by a project manager can be visualized within a Gantt chart.

Critical Path Analysis

Gantt charts can be used for different types of analyzes, one of the most prominent ones being a critical path analysis [Kel61]. In general, the critical path of a project is "the sequence of schedule activities that determines the duration of the project" [PMBOK04], which is the longest path through the whole project. All activities or work units that are on this path are critical because if problems occur while realizing such work units this has a direct effect on the project duration. Problems in work units outside the critical path can potentially be compensated.

3 Software Production

Now that the foundations of software architecture and software project planning have been introduced, software production will be defined. The definition in Section 3.1 is followed by a short explanation of the software production life-cycle in Section 3.2. Section 3.3 introduces software production scenarios. Software production scenarios illustrate how the idea of software production can be adopted in single system engineering, domain engineering, and product line engineering. Section 3.4 introduces a meta-model of software production. The meta-model of software production connects the software architecture meta-model and the software project planning meta-model introduced in Chapter 2. It is the basis to define and measure producibility as it will be introduced in Chapter 4. The introduction of the software production meta-model is followed by an example of software production in Section 3.5 and related work in Section 3.6.

3.1 Definition of Software Production

This thesis is based on the following definition of software production:

Definition Software Production: *Software production is the realization of a software system by creating and assembling the architectural elements defined in the software architecture according to a software production plan. Thereby, software architecture and software production plan have been aligned to each other up-front to be able to meet the production goals and mitigate risks potentially causing project failure.*

Software production per definition addresses the practical problem introduced in Section 1.1.

Software production mimics the production of hard goods.

The idea of software production is based on approaches to produce or manufacture hard goods. As already mentioned in Section 1.4, in the manufacturing industry production is planned thoroughly based on the product design and the available production capability of an organization. Thereby, the production capability is made up of the production processes that can be executed in the organizations facilities and the human resources with their skills. It has been experienced in the manufacturing industry that production planning is not possible without a thorough understanding of the product design and vice versa [Bra98]. In terms of Software Engineering, this means that the software architecture representing the product design and the project plan need to be aligned

to each other early on. Alignment means, for instance, that the structure of a system defined in the architecture somehow supports the realization of the features in the order requested by the customer (see example in Section 1.2). If such an alignment has been accomplished, we can talk of software production. For a precise definition of the alignment of software architecture and production plan we refer to Chapter 4.

Software Production is strongly product-oriented

The key characteristic of software production is its strong product-orientation. Creation and assembly of architectural elements are the dominating activities in software production. Typical work activities in software production of information systems could be, for instance, "Create an architectural element of the type service with the following specification…" or "Assemble service A and service B". Development activities like implementation, generation, or testing are then performed in the context of such production work activities. Alternatively, production work activities can first refer to elements of the requirements, for instance, features of a product or workflows in the case of workflow-based information systems, and define work activities like "Realize feature F" or "Realize workflow W". Defining production work activities to produce features or referring to other concepts in requirements engineering is sometimes closer to the customers' point of view as customers typically request certain features. In this case, a mapping to architectural elements is required to derive production work activities on architectural elements and evaluate the alignment of architecture and production plan.

Set-up supply chains

Because of the product-orientation of software production, organizations have the chance to set-up supply chains providing architectural elements to be integrated into the product as it is done in other engineering disciplines. Experts with respect to certain architectural elements can be involved easily and their expertise can be used. Hence, suppliers in software production are selected to deliver parts of the product (not to take over activities).

Production Perspective supports in detecting additional project risks

Software projects can largely benefit from the idea of planning production based on the product's design, i.e. planning the realization of a software system based on the software architecture. By taking this production perspective, potential problems in the construction phase of a project caused by characteristics of the architecture can be identified. Hence, risks during production which are not covered by a typical process-oriented perspective can be mitigated and improvement potentials can be systematically detected and addressed. If an architectural element is supposed to be changed often during construction and does not provide appropriate extension mechanisms, for instance, this potential production problem is not detected without considering the architecture in combination with the production plan. Too often, software projects are planned without thorough consideration of the product to be built, i.e. without considering the architecture and the characteristics of architectural elements, as already mentioned in Section 1.1. Instead, project

managers heavily rely on the best practices they have selected from process frameworks or other sources that typically do not provide specific guidance for the product at hand.

3.2 Software Production Life-Cycle

Figure 22 provides a high level perspective on the software production life-cycle. Requirements Engineering provides requirements to production planning and architectural design. Production planning and architectural design are performed in close collaboration to achieve the envisioned alignment of architecture and production plan. Architecture and production plan are then handed over to production to create and assemble the architectural elements accordingly. While we call the overall approach software production, production in a closer sense is the phase after production planning and architectural design have been performed. The activities do not necessarily need to be performed strictly sequential. Production planning and architectural design can start as soon as an initial set of requirements is available. Production itself is supposed to be performed in an iterative and incremental way anyway which allows to evolve production plan and architecture over time if required.

Requirements Engineering in Software Production

An interesting aspect is that requirements engineering does not need to be performed completely up-front but can be delayed to a certain degree after architectural design and production planning have been initially done. We know, for instance, that not all requirements are relevant for architectural design. Architectural design and production planning could be performed based on the set of requirements that is relevant for architectural design and production planning.

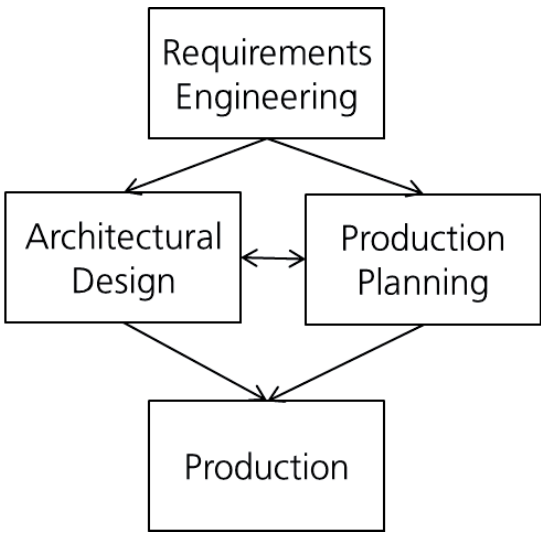


Figure 22: Software Production Life-Cycle

Detailed requirements analysis of further requirements could be performed on demand while production is running and input for creating respective architectural elements is required. Consequently, time could be saved by stronger overlapping requirements engineering, architectural design, production planning, and production. This thesis does not elaborate this idea further. Instead we refer to the work of Adam on tailoring the requirements engineering process for software production [Ada10].

3.3 Software Production Scenarios

Software production requires additional investments in early phases of a software project. Alignment of software architecture and production plan does not happen automatically but requires time and effort that is assumed to be saved later on as production problems are prevented. Nevertheless, organizations might be reluctant regarding the investment in an initial alignment of software architecture and production plan. In this section, we discuss three scenarios where an investment into software production has the highest return on investment.

3.3.1 Single Systems with repeating Production Sequences

Software production can pay off already while developing a single system, especially if certain production sequences are repeated several times throughout the project.

Let's consider a workflow-based information system, for instance, to support certain office workflows. Typically, such workflow-based information systems are constructed workflow by workflow in several iterations for several reasons. First of all, by introducing support for selected workflows quickly organizations can get a positive return on investment earlier. Furthermore, feedback from introducing first workflows can be leveraged in later iterations by all involved stakeholders including users, members of IT departments, etc.

Each iteration follows the same production sequence to realize the selected workflows. Workflows are realized by certain types of architectural elements and certain development activities have to be repeated per architectural element in each iteration. Consequently, aligning the architecture defining how workflows are technically realized and the production plan for single iterations once enables organizations to benefit from this initial investment in ach iteration. One can also think of providing specific guidance and tool support for producing workflows.

3.3.2 Producing similar Systems in a specific Domain

A potentially even higher return on investment than in the scenario described above can be gained if organizations produce similar systems in a specific domain. We assume in this case, that a reference architecture for systems in the respective domain exists.

As mentioned in Section 2.1, "a reference architecture defines element types, allowed interactions, and how the domain functionality is mapped to architectural elements" [HND99]. If systems in a specific domain are based on the respective reference architecture, certain architectural element types need to be produced again and again and an overall approach to produce such a system can be planned and manifested in a kind of domain specific reference production plan. If the reference architecture and the reference production plan have been aligned once, an organization can benefit from this alignment several times.

The example of producing mobile business applications that will be introduced in Section 3.5 refers to this scenario.

3.3.3 Product Line Engineering with a pre-defined Scope

Finally, software production can be adopted in the context of software product line engineering. As mentioned in Section 2.1, product line engineering aims at constructing several products sharing commonalities but also well-defined variabilities based on reuse. Therefore, all product line members share the same product line architecture.

During scoping product candidates to be built are elicited. Based on the scope and the identified commonalities and variabilities a product line infrastructure containing reusable artifacts is developed during family engineering. Here, a production plan for members of the product family can be derived and an alignment with the product line architecture can be established. Consequently, all application engineering projects, i.e. the projects building the concrete products for customers, could benefit from the alignment of product line architecture and production plan that has been initially established during family engineering.

The term production is already used in the product line community. But production in their sense does not yet consequently consider the relationship of architecture and production plan [Car08].

In the following section, software production and the related concepts are subsumed in a software production meta-model.

3.4 Software Production Meta-Model

The software production meta-model connects the meta-models of software architecture, software project plan, organization, and development processes describe in Chapter 2. It provides the basis for a deeper understanding of software production and is the basis to define and measure producibility, as introduced in Chapter 4. Before the integrated meta-model is presented, software production plans as a specialization of project plans are introduced.

3.4.1 Software Production Plans

Software production plans are defined as follows:

Definition Production Plan: *A production plan is a project plan consisting of*

- *a production work breakdown structure,*
- *a production schedule defining production iterations,*
- *an assignment of resources to elements of the production work breakdown structure, and*
- *a description of the production process.*

The definition of a production plan introduces several new concepts that will be defined in the following:

Definition Production Work Breakdown Structure: *A production work breakdown structure (production WBS) is a product-oriented work breakdown structure consisting of production work activities.*

Definition Production Work Activity: *A production work-activity is a product-oriented work activity.*

As already discussed in Section 2.2.2, a product-oriented work breakdown defines work units by referring to elements of the requirements or to architectural elements to structure the work. If elements of the requirements are used, they need to be mapped to architectural elements before the alignment of software architecture and production plan can be evaluated and software production can really start.

The production schedule being part of a production plan defines production iterations. Thereby, production iterations are defined as follows:

Definition Production Iteration: *A production iteration is an iteration adding a well-defined increment to the product in production.*

An increment is defined as follows in this thesis:

Definition Increment: *An increment is a set of new architectural elements or extensions/modifications to architectural elements that are produced and added to a product in one production iteration.*

A production iteration is in that sense a product-oriented iteration as it explicitly adds to the final product. Iterations that would not be product-oriented are requirements or design iterations extending or modifying the respective requirements or design artifacts. A production iteration has a 1:1 mapping to an increment as each production iteration contributes a well-defined set of architectural elements or extensions/modifications to architectural elements.

Assigning work activities to production iterations leads to an overall production schedule. As production iterations are performed sequentially, they introduce a certain order to work activities assigned to different production iterations. A more detailed planning of the order of work activities is then performed inside iterations. It depends on the duration and complexity of production iterations, how much internal planning is required in each case. In general, production planning can be performed in the same way again inside each iteration. In that sense, production planning is a recursive approach.

The assignment of work activities to production iterations should be accompanied by an estimation of effort and duration.

Production plans can include production milestones. A production milestone is defined as follows:

Definition Production Milestone: *A production milestone is a milestone referring to the completion of a certain architectural element*.

Production milestones often refer to aggregated architectural elements like layers, clusters, or sub-systems. Consequently, production milestones would be "Layer L completed" or "Sub-System S completed".

Software production plans refer to production processes. As described in 2.2.4, project planners are responsible for selecting/defining development processes. Consequently, in the case of software production the need to select/define production processes.

In Figure 23, the production plan meta-model is shown.

In the following section, software production processes are introduced.

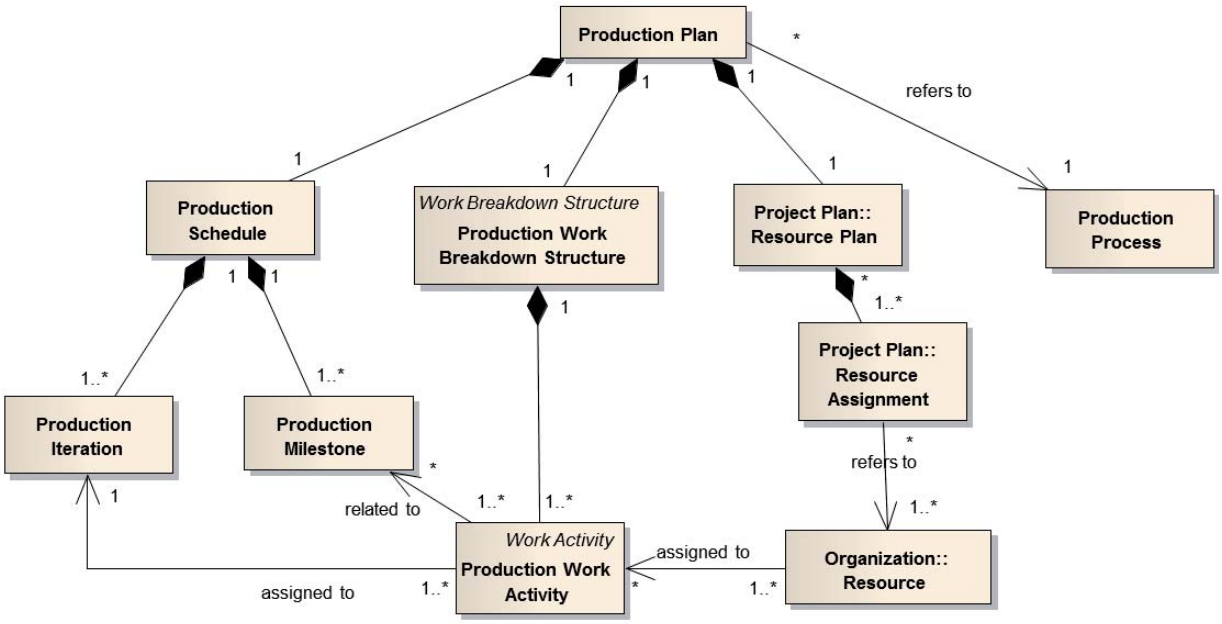


Figure 23: Production Plan Meta-Model

3.4.2 Software Production Processes

Software production processes guide in how to perform the work activities in software production. They have to be product-oriented processes as they are supposed to be applied in the context of a certain architecture, reference architecture, or product line architecture.

Software production processes are defined as follows:

Definition Software Production Process: *A software production process consists of production work activity types describing how to produce architectural element types defined in corresponding architectural styles used in a products architecture, a reference architecture, or a product line architecture.*

Definition Production Work Activity Type: *A production work activity type is a description of a procedure that should be followed to produce a certain architectural element type. A production work activity type can refer to development activity types of a corresponding development process.*

The product-orientation of software production processes becomes apparent in the relationship of production work activity types and architectural element types. Architectural element types are the work products consumed and produced by the production work activities. Hence, a

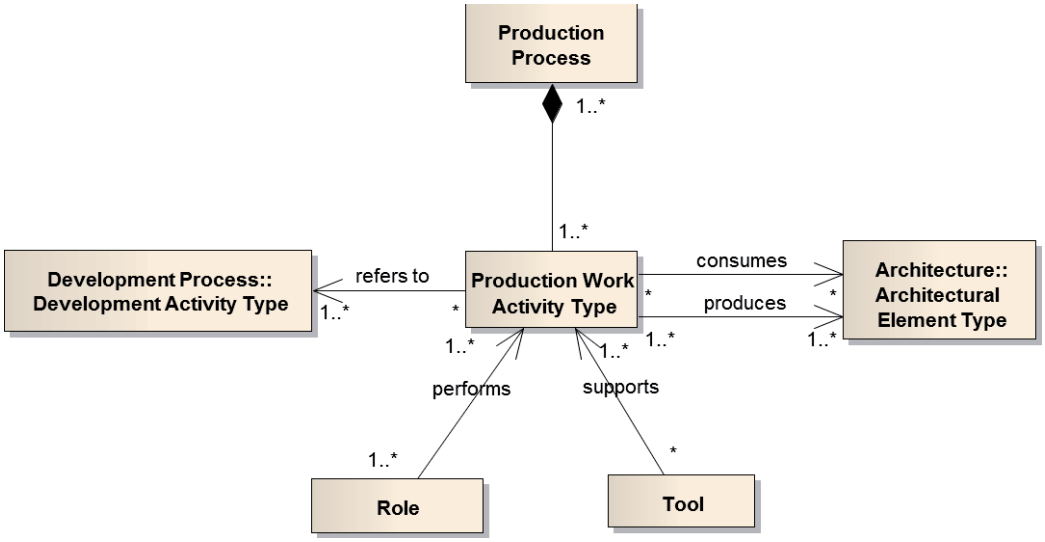


Figure 24: Production Process Meta-Model

production work activity type describes how to produce a certain part of a product.

The following examples illustrate the product-orientation of production processes. Let us assume a system uses a pipe and filter style to process a certain stream of data. Respective production work activity types would be in this case "Create Architectural Element of Type Pipe" and "Create Architectural Element of Type Filter". The production work activity types would describe a procedure on how to produce pipes and filters. This could include, for instance, a description how to internally design a pipe or a filter, how to implement them by means of a certain technology, how to test them, etc.

This example also illustrates the relationship of a production work activity types and development activity types. A production work activity types refers to development activity types generally describing how to design, implement, or test. While production work activity types and development activity types are both describing parts of processes, there are certain major differences:

Production activity types are applied to architectural element types of a specific architectural style, reference architecture, or product line architecture, i.e., they are not generally applicable to any system with any architecture. Development activity types are much more general and can be applied to a huge number of different system with different architectures. This means that production work activity types provide more specific guidance on how to produce a certain system than development activity types.

3.4.3 Integrated Software Production Meta-Model

The integrated meta-model of software production connects the meta-models of architecture, production plan, production process, and organization introduced above. As we will see later, the production meta-model is the basis for the quality model of producibility introduced in Chapter 4.

With respect to the alignment of software architecture and software project plans that is envisioned in this thesis, the connection of the software architecture meta-model to the production plan meta-model is an essential part of the integrated production meta-model. Figure 25 shows the Software Production Meta Model.

Connecting Production Work Activities and Architectural Elements

The key connection between the production plan meta-model and the architecture meta-model is established between production work activities and architectural elements. Production work activities consume and produce architectural elements. Production of architectural elements can mean in this case creation, extension, or modification. Production work activities do not necessarily need to consume other architectural elements, they can create architectural elements from scratch. The relationship of production work activities and architectural elements represents the key characteristic of software production of being architecture-centric. Production work activities always refer to architectural elements that are produced by them. Hence, each production work activity directly contributes to the product and adds at least a small part to it.

Connecting Production Work Activities, Increments, and Architectural Elements

Production work activities are assigned to production iterations similar to how work activities are assigned to iterations in the project plan meta-model. We call the concept production iteration in this case to again emphasize the product orientation in this case. A production iteration produces an increment of the product which is a concrete extension of the overall product, i.e. architectural elements are added, extended, or modified. Iterations of a project do not necessarily need to add to the final product, they could also be iterations to refine the requirements or refine the design of certain architectural elements, but no direct effect on the product would exist in this case as no code is produced. By assigning production work activities to production iterations a link between architectural elements and production iterations is established that is made explicit in the meta-model by the "involved in" relationship between them.

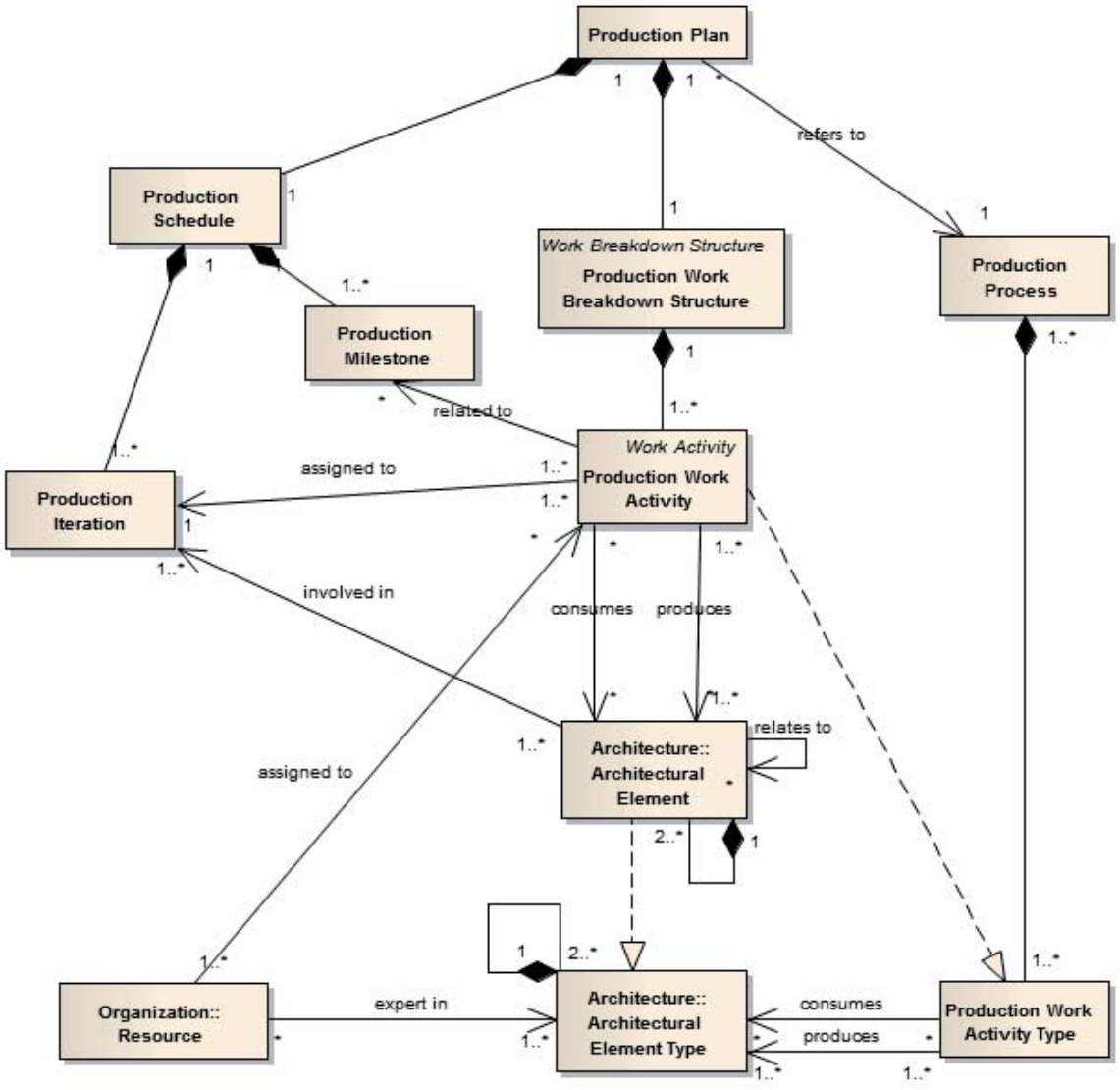


Figure 25: Software Production Meta-Model

Aggregated architectural elements and Production Iterations

A specific aspect of interest is the question how aggregated architectural elements like layers, clusters, or subsystems are related to production work activities and production iterations. Aggregated architectural elements can be used to define higher level production work activities like "Produce Layer L" or "Produce Cluster C" and structure the work accordingly. Even if this is not the case, aggregated architectural elements are related to production iterations as architectural elements being part of them are produced in certain production iterations.

Production Work Activity Types and Architectural Element Types

A similar relationship than between production work activities and architectural elements exists between production work activity types and architectural element types. Production work activity types are part of the production process and consume as well as produce architectural element types. Via production work activity types and architectural element types the production process thus is related to a software reference architecture, software product line architecture, or architectural style. This represents one of the key characteristics of production processes to be product-specific.

Resource Assignment in Software Production

Resources are assigned to production work activities. In an ideal case, resources are experts with respect to certain architectural element types and can be assigned to production work activities that create architectural elements of such types. As a consequence of adopting a software production approach, organizations should train their staff in producing certain architectural elements types. They could even set up departments which combine people that are experts with respect to the same architectural element types to give them the chance to evolve their competences together.

The following section discusses an example of software production.

3.5 Software Production Example

The following example aims at further clarification of the ideas of software production and the software production meta-model. The domain we selected are mobile business applications. Organizations can use mobile business apps to offer their business services via mobile devices like smartphones or tablets or to provide mobile services supporting the workflows of their employees. We introduce a simple reference architecture for mobile business applications and sketch an excerpt of a related production process.

3.5.1 Reference Architecture for Mobile Business Apps

Mobile business apps are software products deployed on mobile devices like smartphones or tablets that are typically put on top of an existing IT infrastructure consisting of several backend systems. A travel management app as mentioned above must be connected to the backend systems of potentially different organizations offering flights, hotels, and rental cars.

Different architectural alternatives can be selected for mobile business applications from so-called native applications, via hybrid applications to web-applications. In our example, we aim at producing a native application for the iOS platform [iOS], i.e. an application for iPhone or iPad.

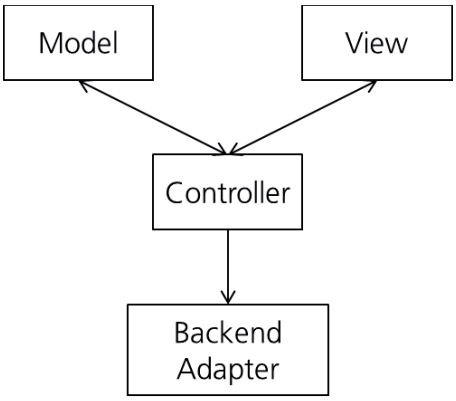


Figure 26: Example Reference Architecture

Mobile business apps typically leverage the Model-View-Controller (MVC) pattern [GFJ+94]. Most mobile development platforms like iOS, Android [And], or Windows Phone 7 [WP7] provide a specific implementation of the MVC pattern.

Based on the characteristics mentioned above the following simple reference architecture shown in Figure 26 is chosen for mobile business apps in this example.

The core of the mobile business app is realized according to the MVC pattern. The connection to backend systems is realized via so-called back end adapters. The role of the backend adapters is to abstract from technical details of the respective backend systems from the point of view of the mobile business app and map the data model of the mobile business app to the data model of the backend systems.

Based on our existing experience in producing mobile business applications we can make additional assumptions regarding the reference architecture. We assume that backends offer their service via web services [W3C04] and that data objects are transferred by means of JSON [JSON].

The reference architecture is simplified as certain important aspects for mobile business apps are not considered in the example.

If we apply our software production meta-model to the reference architecture, we can identify a certain set of architectural elements types, namely: Model, View, Controller, Backend Adapter, and Backend Service.

3.5.2 Development Process vs. Production Process

The production process must provide support to produce the architectural element types defined in the reference architecture, i.e. production work activity types. We select the architectural element type “Backend

Adapter" in this case and define a production work activity type "Produce Backend Adapter". The production work activity type "Produce Backend adapter" is supposed to provide specific guidance on how to produce backend adapters for mobile business applications.

As a basis for defining the production work activity types we decide to use a development process based on the Open Unified Process (OpenUp) [OpenUp], an open source version of the Rational Unified Process (RUP) [Kru03]. This means in terms of our software production meta-model that development activity types are selected from the OpenUp.

Table 1 shows the selected development activity types and the production work activity type "Produce Backend Adapter" as one example. The production work activity type is based on the development activity types as it follows the general sequence of development activity types specified in OpenUp. The major difference is that each step in the production work activity type is product-oriented as it provides concrete guidance on how to perform the respective steps for backend adapters and not for any kind of architectural element.

The production work activity type "Produce Backend Adapter" contains one additional step "Deploy and simulate integrated mobile business app" due to the specific product respectively the mobile development platform to be used. Such product specific steps cannot be derived by selecting development activity types from existing process frameworks. The mobile business app is deployed on the iPhone or iPad simulator that is available in this case to test the app before deploying it to the physical mobile device. This gives the developers the chance, for instance, to check for memory leaks which is a known problem in iOS applications and that can be detected by means of the Instruments tool.

#	**OpenUp based Development Activity Types**	**Production Work Activity Type** "Produce Backend Adapter"
1	*Design the Solution:* Identify the elements and devise the interactions, behavior, relations, and data necessary to realize some functionality.	*Design Backend Adapter:* Model the relations between the backend data model and the data model of the mobile business app. Model the relationship of the backend adapter to the backend services. Model the behavior of the backend adapter. Consider minimizing the number of calls of backend services to eventually save cost. …
2	*Implement Developer Test:* Implement one or more tests that enable the validation of the individual implementation elements through execution.	*Implement Backend Adapter Test*: Implement one test per public method and equivalence class of the backend adapter. …
4	*Implement Solution:* Implement source code to provide new functionality or fix defects.	*Implement the backend adapter:* Implement the mapping of the mobile business app data model to the backend data model. Implement the behavior as specified in the backend adapter design. Implement the communication with the backend services using restful web services. Use JSON to exchange data between backend adapter and backend services. …
5	*Run Developer Test:* Run tests against the individual implementation ele-	*Run Developer Tests:* Run the developer tests of the backend adapter using

	ments to verify that their internal structures work as specified.	OC Unit …
6	*Integrate and Create Build:* This task describes how to integrate all changes made by developers into the code base and perform the minimal testing to validate the build.	*Integrate and Create Build:* Integrate the backend adapter with the other architectural elements already created. …
		Deploy and simulate integrated mobile business app: Run integrated mobile business app on the iPhone Simulator Use the instruments tool to detect potential memory leaks …

Table 1: Example - Development Process vs. Production Process

Table 1 shows only an excerpt of the overall production process. For each architectural element type of the related reference architecture as production work activity type would need to be defined that provides specific guidance.

3.6 Related Work

In this section, an overview on related work on software production is provided.

Agile Software Engineering

Agile Software Engineering is an approach claiming to be essentially product-oriented. The Agile Manifesto [AM01], for instance, argues for product-orientation especially by mentioning the following three (out of twelve) principles:

– "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

– "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."
– "Working software is the primary measure of progress."

Agile projects are largely driven by the customer as the customer decides on the scope of the project and the next releases and iterations [Hun06].

Although agile approaches to Software Engineering are essentially product-oriented, software architecture does not explicitly play a central role in an agile project. Big-design up-front is supposed to be to heavy-weight and causes overhead effort if architects plan for scenarios that will potentially never get relevant during the project as requirements continuously change anyway. Hence, agile Software Engineering argues for lightweight and emergent design [BA04] [Amb02] [CLM+06]. Because of the continuous change, agile software engineering also does not perform detailed planning for a whole project but only for upcoming iterations or releases with an almost fixed scope. In Scrum [Coh09], requirements or tasks to be performed are collected in a so-called backlog an pulled from there over time. From a design and planning perspective, agile Software Engineering differs from software production. Software production requires more architectural design and production planning up-front to be able to align architecture and production plans to each other and reduce risks related to a potential misalignment.

Combining agile principles and software production might be an interesting combination to be considered. While agile thinking could force organizations adopting a software production approach to carefully think about the architectural decisions and production planning decisions that really need to be taken up-front, agile approaches could more explicitly consider the role of the architecture in general and specifically during planning.

Lean Software Engineering

Lean Software Engineering [PP03] is an approach that is often referred to as an agile approach. We mention it here separately, because it explicitly refers to the principles of lean production in other engineering disciplines. The most prominent example on lean production is the Toyota Production System (TPS) [OB88]. The TPS adopts principles of Kanban, one of the most prominent ones being "Eliminate Waste".

Hence, they adopted, for instance, lean principles to software development. While lean Software Engineering also takes a production perspective and try to remove inefficiencies, etc. it does not explicitly consider the role of the architecture. It rather focuses on the development processes and how to optimize them which could be complementary to the architecture-centric approach chosen in this thesis.

Software Factories

Software factories [GSC+04] envision an industrialization of software engineering increasing productivity and achieving similar maturity levels than manufacturing industries. Development by assembly is one of the key ideas of software factories similar to software production. Software supply chains involving suppliers in the development process are set-up and managed. Development by assembly and introduction of supply chains move software development closer to the state of the practice in other industries. Products are developed based on reuse according to a product line approach. Software factory schemas describe the artifacts that must be developed to produce a product of a product family. Furthermore, the relationships between such artifacts and transformation rules to derive certain artifacts from others are part of the software factory schema. Software factory templates are implementations of software factory schemas that can be loaded into tools to support the developers in building products. Recurring, menial development tasks are automated to focus more on the creative tasks.

Software Product Line Engineering

As mentioned above, software product line engineering is concerned with the construction of products sharing commonalities and well-defined variabilities based on reuse from a product line infrastructure. The so-called product line life-cycle shown in Figure 27 illustrates this. Family Engineering (FE) creates reusable artifacts based on a pre-defined scope. Application Engineering (AE) produces products for the customers of a product line organization based on reuse. The construction of members of the product line during AE is called production in certain cases [CN02] [McG04].

Krueger talks of so-called software production lines [Krue01] [Krue02]. Thereby, a production line is a specific type of product line. In a production line, the reusable artifacts are under configuration management in the product line infrastructure. Products are produced by means of a tool infrastructure based on the reusable artifacts. This leads to a high degree of automation of the application engineering process. The individual products for the customers are not under configuration management, but are produced on demand. Changes are always applied to the reusable artifacts and products are reproduced if required based on the changed reusable artifacts.

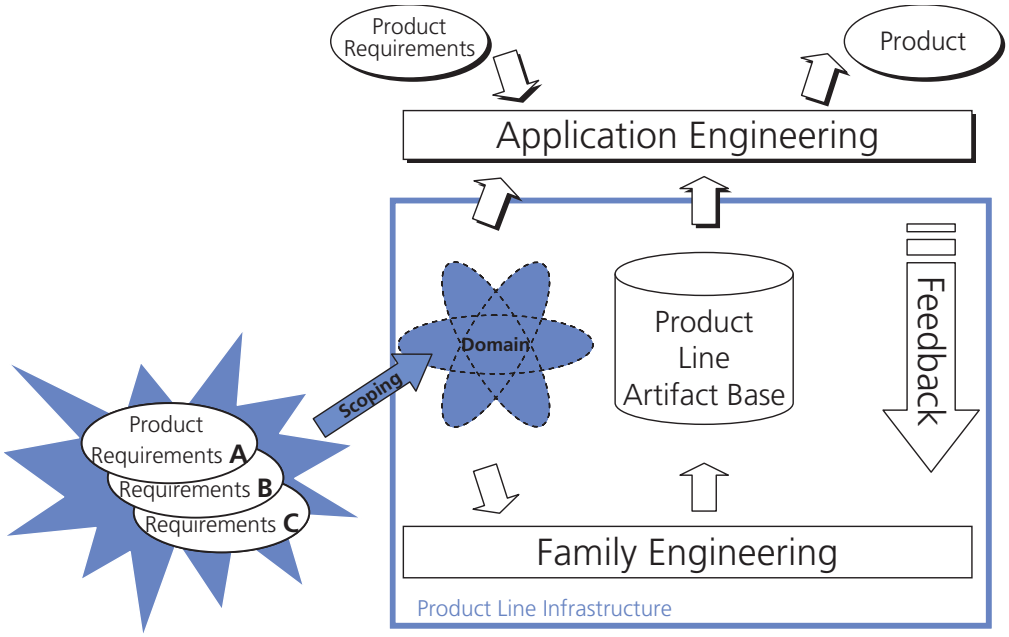


Figure 27: Product Line Life-Cycle

Production planning is mentioned as an important activity in software product line engineering [MC08]. Thereby, "a production plan is a description of how core assets are to be used to develop a product in a product line" [CM02]. It provides guidelines on how to perform certain work activities during AE or production. In [McG04], McGregor argues for setting-up a production system for a software product line similar to production environments in manufacturing that enable the production of members of a product line according to the production plan.

The architecture is often referred to as the key artifact in software product line engineering enabling reuse, as all members of the product line share the same architecture. Nevertheless, production planning in software product line engineering does not systematically consider the product line architecture or even provide feedback to architects to change the architecture to be better aligned with the production plan [Car08].

The Term Production in the Context of Software

The term production is used by software developers for several other purposes in combination with software that should not be mixed up with our definition.

The process of creating an executable piece of software out of source code is sometimes called production. This process is typically highly automated by means of build scripts.

The distribution of software after it has been developed is sometimes called production. Software is copied in this case to be distributed to many customers. But in contrast to hard goods manufacturing copying of software causes almost no costs.

The operation of software, for instance, in a data center, or the execution of software are sometimes also called production.

4 Quality Model of Producibility

This chapter introduces producibility as the quality attribute in the focus of this thesis. The definition is based on the software production meta-model introduced in the previous chapter. Section 4.1 gives the general definition of producibility. The following sections then introduce producibility metrics that can be applied by architects and production planners. The producibility metrics are complemented by a set of context factors that must be carefully considered while interpreting the values of the producibility metrics.

4.1 Definition of Producibility

In this thesis, producibility is defined as follows:

Definition Producibility: *Producibility of a software system is the degree of alignment of a system's architecture with the production plan in a given context.*

Thereby, producibility, i.e. the alignment of architecture and production plan, is measured on three dimensions:

- Alignment of architecture and production work breakdown structure
- Alignment of architecture and production schedule
- Alignment of architecture and resource assignments

Figure 28 illustrates the three dimensions of producibility.

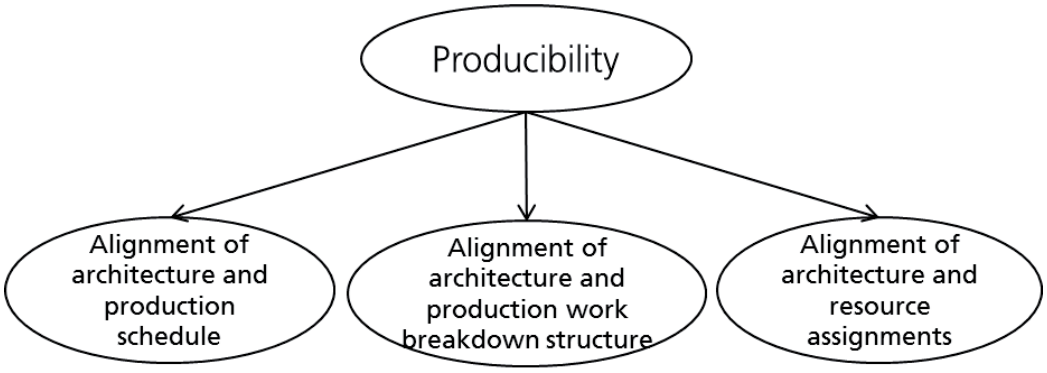


Figure 28: Three Dimensions of Producibility

By means of the software production meta-model introduced in Section 3.4, it is possible to concretize the definition of producibility in each dimension and systematically derive producibility metrics. Figure 29 shows the excerpt of the software production meta-model relevant in this case and the relationship to the three dimensions mentioned above. The architecture is represented by architectural elements, the production work breakdown structure by production work activities. Consequently, the alignment of architecture and production work breakdown structure can be determined based on the relationship of architecture elements and production work activities in a concrete case. In a similar way, the alignment of architecture and production schedule is determined based on the relationships of architectural elements and production iterations. The alignment of architecture and resource assignments is based on the relationship of architectural elements and resources assigned to production work activities respectively architectural elements. Hence, based on the meta-model of software production introduced in 3.4, alignment or misalignment of architecture and project plan can be defined precisely, i.e. in a measurable form.

During measurement, the perspective of the observer is always relevant [BCR94b]. In the case of producibility, the perspectives of architects and production planners are specifically supported by respective metrics. As shown in Figure 30, architects have a specific perspective as their starting point for any considerations are architectural elements. Hence, they are interested in measures relating architectural elements to production work activities, iterations, and resources. The other way round, production planners are primarily interested in measures relating production work activities, iterations and resources to architectural elements.

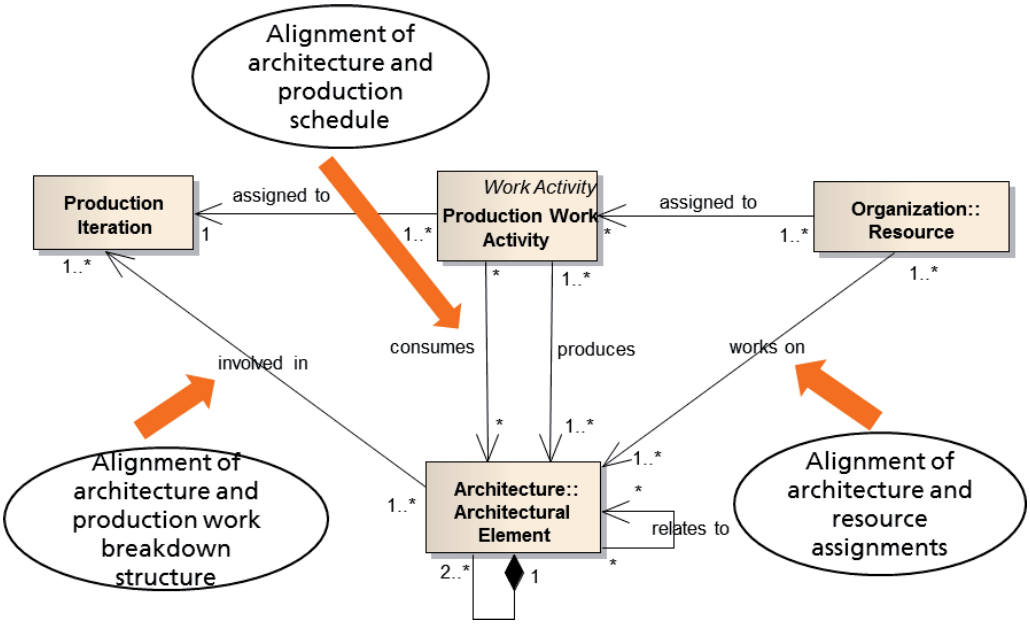


Figure 29: Producibility in the Software Production Meta-Model

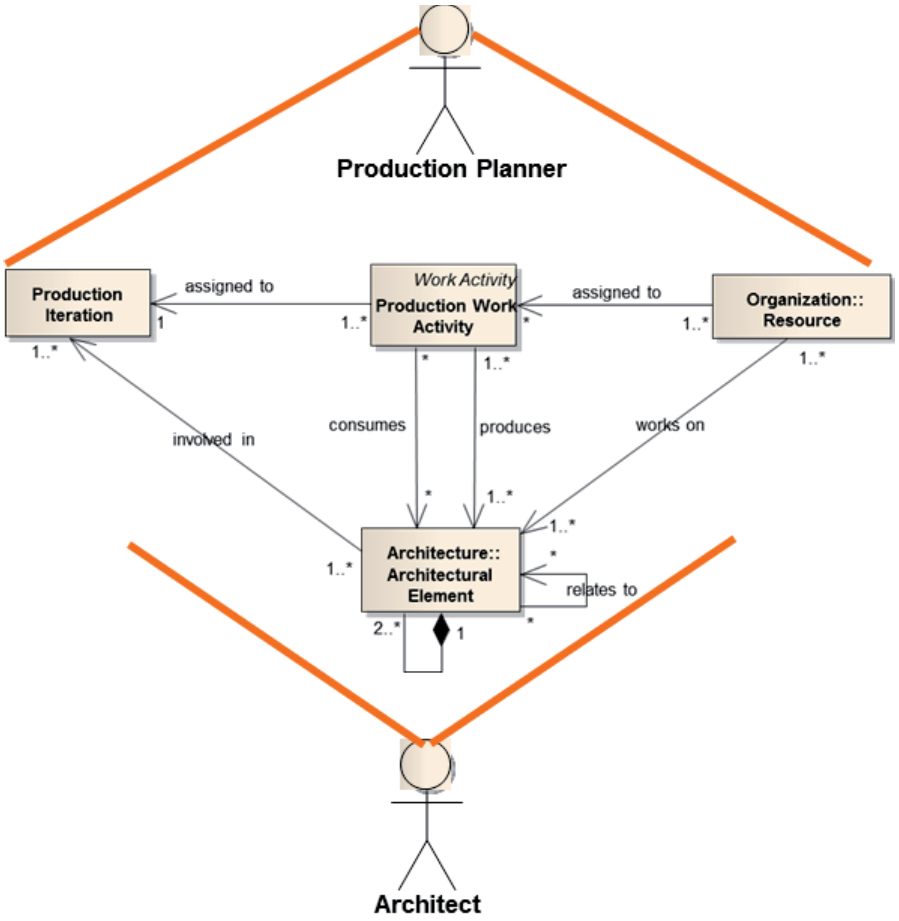


Figure 30: Different Perspectives on Producibility

Producibility and Software Production

The definition of producibility is oriented at the definition of software production given in Section 3.1. We assume that an organization or project team aims at executing a project conform to the ideas of software production. Producibility is then a quality attribute measuring if software production is adopted and the expected effects of software production can be achieved. As mentioned in Chapter 1, the misalignment of architecture and project plan is a major problem in today's projects that is addressed in software production by definition. If positive values of producibility are achieved, it is assumed that projects are more successful, i.e. less budget and time overruns, higher quality of the resulting products, etc. can be achieved. One could also argue that producibility aims at reducing production risks.

In the following section, the alignment of architecture and production work breakdown structure as one sub-attribute of producibility is discussed in detail and concrete metrics are introduced. Before the concrete metrics are defined, we define the following sets of elements:

Definition Set of all Architectural Elements AE_{all}: *AE_{all} is the set of all architectural elements that have been designed into the architecture of a system.*

Definition Power Set of all Architectural Elements $P(AE_{all})$: *$P(AE_{all})$ is the power set of the set of all architectural elements AE_{all}.*

Definition Set of all Production Work Activities PWA_{all}: *PWA_{all} is the set of all production work activities that have been planned in the production WBS of a project.*

Definition Set of Production Iterations PI_{all}: *PI_{all} is the set of all production iterations defined into the production schedule of a project.*

Definition Set of assigned Resources RES_{all}: *RES_{all} is the set of all resources that have been assigned to production work activities in a production plan.*

4.2 Alignment of Architecture and Production WBS

This section introduces how to determine and measure the alignment of architecture and production work breakdown structure via the relationship of architectural elements (AE) and production work activities (PWA). Hence, we focus on the excerpt of the software production meta-model shown in Figure 31.

Required Artifacts

The measurement of the alignment of architecture and production work

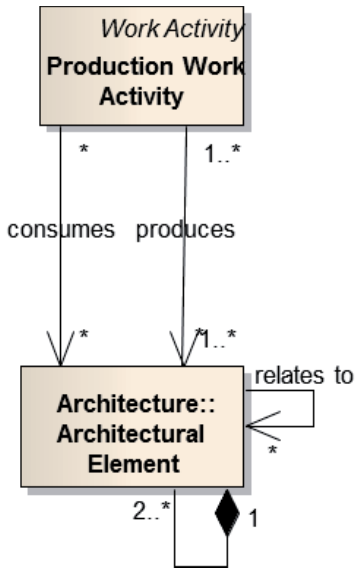


Figure 31: Focus of Alignment of Architecture and Production WBS

breakdown structure is based on several artifacts that need to be available as a prerequisite. A production work breakdown structure is required, i.e. it either defines work activities based on features or directly refers to architectural elements (see Section 2.2.2). If the production WBS is based on features, an artifact containing traceability information from features to architectural elements is required. If such an artifact is not available, the traceability information must be derived, for instance, based on interviews with requirements engineers, architects and developers, which can be a time consuming and effort intensive task. Furthermore, a structural view of the architecture is required, i.e. a view showing architectural elements and their relationships. Typical architectural views visualizing structural information are a module or a component and connector view. Both are appropriate in this case, as both are architectural elements according to our architecture meta-model presented in Section 2.1. We do not consider the differences between modules and components to be relevant at this point.

In the following, measures directly characterizing the relationship of architectural elements and production work activities are introduced. As mentioned above, we distinguish the two perspectives of architect and production planner as each measure is of specific interest to one of these two roles.

4.2.1 Architect's Perspective

The architect is primarily interested in the relationship of architectural elements and production work activities. For each single architectural element defined in the architecture, the relationship to production work activities needs to be considered. Hence, the following metrics are defined per architectural element.

Metric Definition: *AE_{all}->N: #PWA_producing(AE) = n, where n is the number of production work activities involved in producing the architectural element AE.*

The notation used for describing the metric must be interpreted as follows: The metric #PWA_producing is a function assigning each architectural element AE of the set AE_{all} a natural number n out N, i.e. the set of all natural numbers.

Metric Definition: *AE_{all}->N: #PWA_consuming(AE) = n, where n is the number of production work activities that are consuming the architectural element AE.*

From an architectural perspective, a production work activity PWA1 is consuming an architectural element AE2 that is produced by PWA2, if an architectural element AE1 produced by PWA1 has an architectural relationship to AE2.

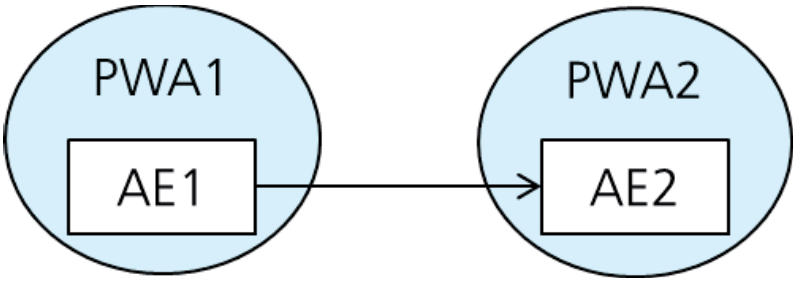


Figure 32: Example Metrics AE and PWAs

The two metrics are illustrated in Figure 32. AE1 is in this case produced by one production work activity PWA1, AE2 by one production work activity PWA2. AE2 is consumed by one PWA. Hence, the resulting values are:

#PWA_producing(AE1) = 1

#PWA_producing(AE2) = 1

#PWA_consuming(AE2) = 1

Many PWAs modifying an AE lead to a huge number of changes applied to the AE over time. Hence, the AE can get a so-called hot spot. Hotspots are a potential source of entering defects, as potentially various developers change it, conflicts arise, etc. In that sense, an AE that is produced by many AEs is critical with respect to quality but also delays, and exceeding of planned effort. In an ideal case, an AE is only modified by one PWA, i.e. #PWA_producing(AE) would have a value of 1 as in the example above.

A high number of PWAs consuming an AE points out the relevance of the AE with respect to the overall product. If production problems arise during production of the AE being it delays or defects the success of all the PWAs consuming the AE is jeopardized. If the AE is not finished in time, consuming PWAs can also be delayed. If the AE has poor quality and is used in many follow-up PWAs the quality of their results is potentially also harmed. Hence, an AE with a high value for #PWA_consuming(AE) should be marked as critical. In an ideal case, #PWA_consuming(AE) = 0. But this cannot be a goal to be achieved for all architectural elements. If all AEs would have a value of zero for #PWA_consuming(AE), this would mean that either the whole product would be produced in one big step, i.e. one PWA, or the architectural elements produced are not connected in any way, which would mean that they do no longer form a system.

So far, single architectural elements and their relationships to production work activities have been considered. But architectural elements are always related to other architectural elements, as it is modeled in the ar-

chitecture meta-model. It is important to have a look on how cohesive sets of architectural elements, i.e. architectural elements being part of an aggregated architectural element like a layer, cluster, or subsystem, or architectural elements that are coupled for other reasons, and production work activities are related. This is important for several reasons:

- The overall behavior of a system is realized by the interaction of architectural elements, for instance, a set of components communicating with each other. Let us consider the example introduced in Section 3.5 again. A backend adapter in combination with a cache component could together be the basis to realize offline working with the travel management app. If offline work is supposed to be a critical behavior of the system, the question is if these two architectural elements should be realized in the same production work activity to make sure that they function together as supposed and the offline functionality can be tested as part of one production work activity.
- Certain architectural elements are supposed to be realized in a uniform way. Using the example of the travel management app again, all backend adapters should be realized uniformly. Furthermore, they are assigned to the same layer. If the uniform realization of the overall layer is supposed to be critical and not easy to achieve in the given project context, the backend adapters maybe should be realized in one or at least closely related production work activities.

Figure 33 and Figure 34 illustrate the problem. A system consists of three layers and the architectural elements AE1, AE2,… AE11. The bottom layer is supposed to be critical for several reasons. Each architectural element, i.e. A8, A9, A10, and A11, is supposed to be realized in a uniform way. If we assume this layer to be a backend integration layer as mentioned above, an architectural requirement could be that each architectural element minimizes the number of transactions to the backend to save cost as each transaction is billed. This requires a certain expert knowledge that should be shared across all the production work activities involved in producing the architectural elements of the backend integration layer. In Figure 33, four production work activities produce the architectural elements of the bottom layer. If the knowledge on how to uniformly produce the architectural elements is not shared appropriately among the four production work activities, there is a certain risk that the requirement will not be fulfilled.

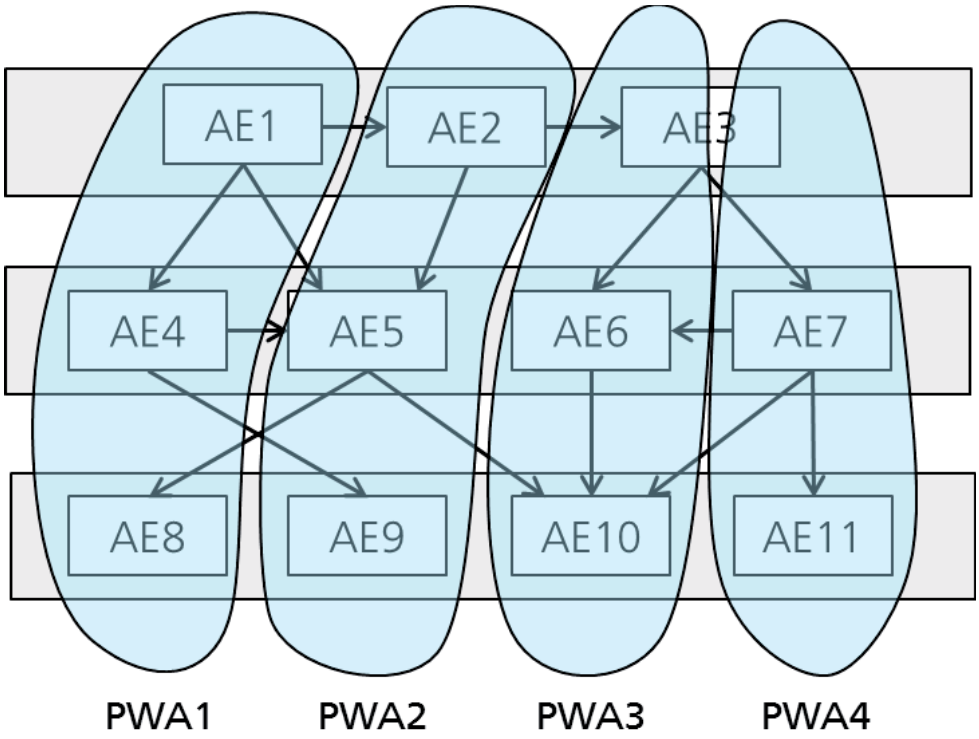


Figure 33: Example 1 - Number of PWAs producing Set of AEs

Figure 34 provides an alternative solution. The architectural elements of the bottom layer are produced in one production work activity which reduces the risk of not fulfilling the requirement of uniform production in this case. Another reason for preferring the solution shown in Figure 34 could be that an organization wants to outsource the production of the bottom layer to an external supplier. In this case, it would be better to outsource one production work activity instead of involving an external supplier into four production work activities.

The following metric is capable of expressing the fact discussed in the example above:

Metric Definition: *$P(AE_{all})$->N: #PWA_producing({AE}) =n, where n is the number of production work activities producing the set {AE} of architectural elements.*

The metric expresses the fact that the production of a set of architectural elements that is cohesive for some reason should not be spread over many production work activities. If the production work activities split the set of cohesive architectural elements, the correct realization of the requirement causing the cohesion might be jeopardized.

In the following sub-section, the production planner's perspective on the relationship of architectural elements and production work activities is presented.

4.2.2 Production Planner's Perspective

The production planner is primarily interested in the relationship of production work activities and architectural elements. For each single production work activity defined in the production work breakdown structure, the relationship to architectural elements needs to be considered.

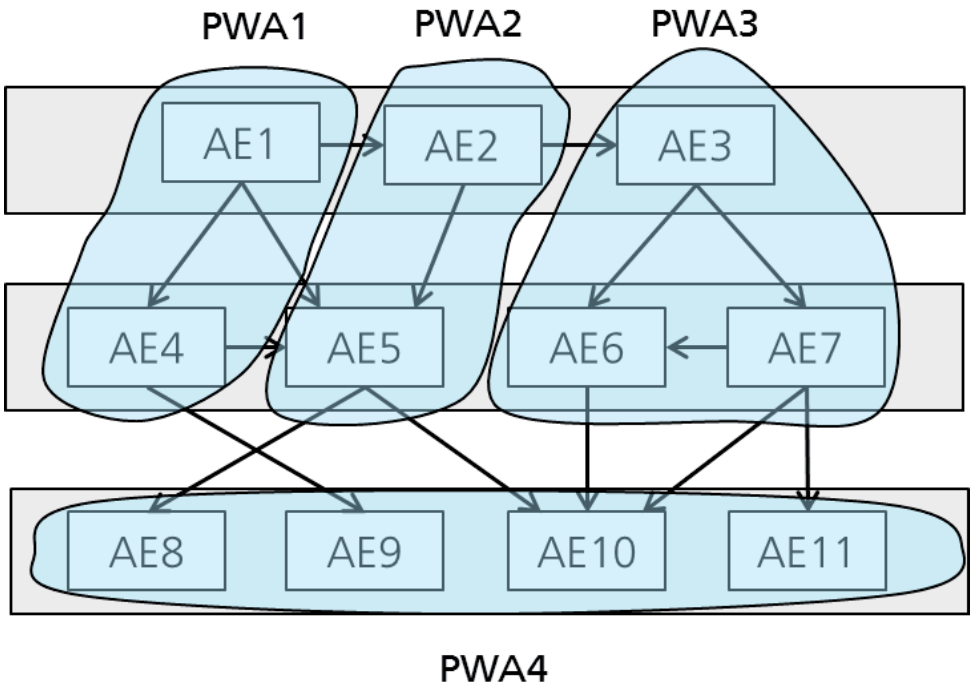


Figure 34: Example 2 - Number of PWAs producing Set of AEs

Hence, the following metrics are defined per production work activity.

Metric Definition: *PWA_{all}->N: #AE_produced_by(PWA) = n, where n is the number of architectural elements that are produced by the production work activity PWA.*

Metric Definition: *PWA_{all}->N: #AE_consumed_by(PWA) = n, where n is the number of architectural elements that are consumed by the production work activity PWA.*

A high number of AEs modified by a PWA is a first indicator for criticality of the PWA with respect to producibility. Modifying various AEs requires knowledge on a potentially large part of the architecture. The modifications made by the PWA are in that sense not necessarily local. Hence, the WA is a potential source of delays, increased effort, or defects. In an ideal case, #AE_produced_by(PWA) is one.

A large number of AEs consumed by a PWA indicates that a large number of prerequisites needs to be fulfilled to start the PWA, i.e. a large number of AEs must have been produced before. Hence, a high value for #AE_consumed_by(PWA) indicates that a PWA is critical in the sense that its start can be delayed because of missing inputs and that potential follow-up PWAs will also be delayed. One could argue that it is enough if the specifications of consumed AEs are known to be able to start a PWA. But experience shows that various details of specifications of AEs can change during production which suggests to produce them in a certain order. In an optimal case, #AE_consumed_by(PWA) is 0 for a production work activity as this means that it can be performed independent of other PWAs.

The two metrics are illustrated in Figure 35. The production work activity PWA1 produces two architectural elements AE1 and AE2. Furthermore, it consumes two architectural elements AE3 and AE4. Consequently, the resulting values for the two metrics are:

#AE_produced_by(PWA1) = 2

#AE_consumed_by(PWA1) = 2

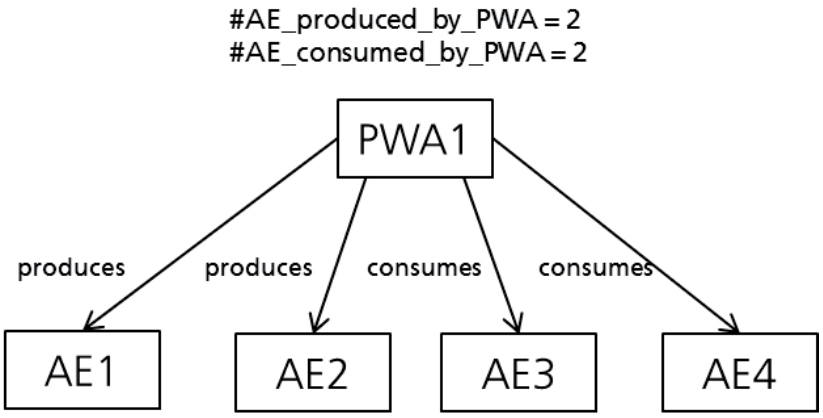


Figure 35: Example Metrics Production Work Activities

Production work activities are related to each other as architectural elements are. Hence, similar to the situation described in Section 4.2.1 in the case of architectural elements, we have to consider such relationships with respect to producibility. Thereby, we specifically consider the relationships between production work activities that are caused by the architectural elements involved.

Relationships between architectural elements cause relationships between production work activities. This fact is illustrated in Figure 36. Figure 36 shows three production work activities PWA1, PWA2, and PWA3. Their relationship becomes visible as soon as we add information on the architectural elements involved in the PWAs. Three relations can be identified between PWA1 and PWA2, and two between PWA2 and PWA3, because the involved architectural elements are related respectively. Such relationships influence the producibility, i.e. in this case the alignment of architecture and production work breakdown structure. Architectural el-

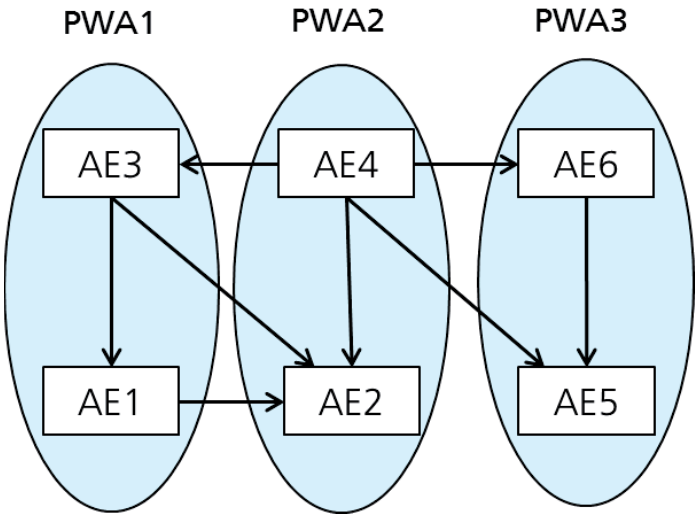


Figure 36: Relationships between PWAs

ements of PWA1 rely on architectural elements of PWA2 and vice versa. Hence, PWA1 and PWA2 cannot easily be produced independent of each other.

One could argue, that the interfaces of AE1, AE2, AE3, and AE4 must be specified up-front, then PWA1 and PWA2 can be performed independent of each other. But experience has shown that even if interfaces have been specified up-front carefully, they most likely change to a certain degree during production. This requires communication between the teams performing PWA1 and PWA2 to maintain consistency, which needs to be considered during production planning.

The fact that PWAs can be coupled via the architectural elements they produce and consume can be measured by means of the following metric:

Metric Definition: *PWA_{all}->N: Coupling(PWA) = n where n is the coupling between the production work activity PWA with any other PWA in PWA_{all} determined based on coupling of architectural elements produced and consumed by PWA.*

In the example shown in Figure 37, we can determine:

Coupling(PWA1) = 2

Coupling(PWA2) = 2

Coupling(PWA3) = 2.

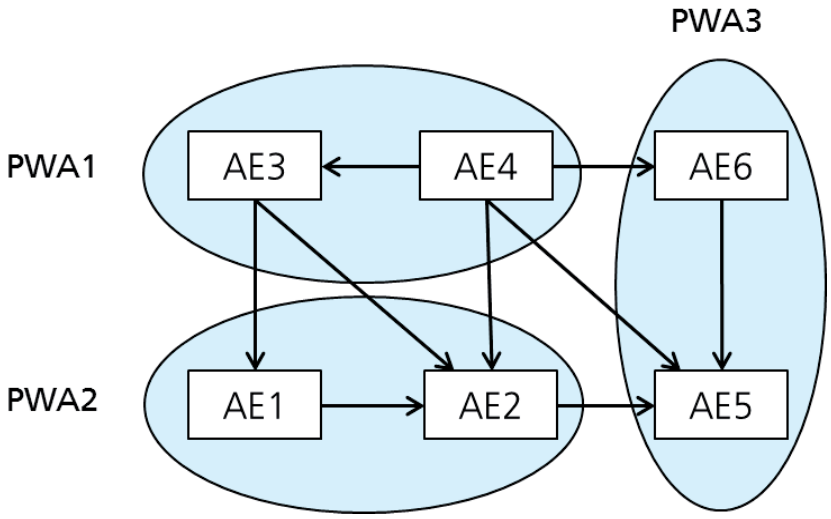


Figure 37: Coupling between PWAs

This means that each PWA is related to each other PWA in this example, which is not a good situation.

Besides coupling between production work activities overlapping can be measured. Overlapping between production work activities means that the sets of architectural elements produced by the production work activities intersect. We also call this sharing of architectural elements in this thesis. The following metric expresses this fact:

Metric Definition: *PWA_{all}->N: #Shared_AE(PWA) = n, where n is the number of architectural elements shared by PWA with any other PWA out of PWA_{all}.*

In the example in Figure 38, we can determine:

#Shared_AE(PWA1) = 2

#Shared_AE(PWA2) = 4

#Shared_AE(PWA3) = 2

If production work activities share a large number of architectural elements, this can be a source of production problems that should be further considered. If the production work activities modify the same parts of the shared architectural elements, for instance, this can lead to conflicts. Sharing of production work activities can be an indicator to better split architectural elements or to change the production work activities and let architectural elements be produced by only one of them. As

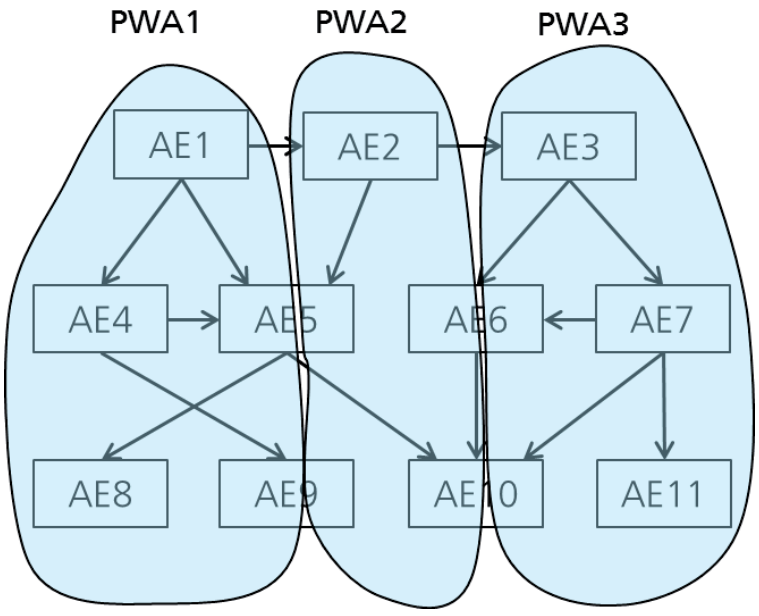


Figure 38: Example sharing of AEs between PWAs

#Shared_AE(PWA) does not take into account the overall number of architectural elements produced by a production work activity, the following metric is introduced in addition:

Metric Definition: *PWA_{all}->R_0^+: %Shared_AE(PWA) = n where n is the percentage of the overall number of AEs produced by PWA that is shared with any other PWA in PWA_{all}.*

If we also consider the overall number of AEs produced by each PWA in the example in Figure 38, we can determine:

%Shared_AE_PWA (PWA2) = 40

%Shared_AE_PWA (PWA2) = 80

%Shared_AE_PWA (PWA2) = 40

This section introduced metrics to characterize the relationship of AEs and PWAs. Such metrics are the basis for all further considerations on producibility as they measure the alignment of architecture and the production work activity, which is fundamental for all further production planning activities. They can be used already in an early phase of production planning and architectural design as they only require first versions of the structure of the system and a production work breakdown structure.

In the next section, the production schedule will be considered with respect to producibility. Hence, in addition to architectural elements and production work activities, iterations are taken into account. Scheduling decisions, i.e., for instance, assigning production work activities to iterations, can compensate potential problems indicated by the values of the metrics introduced in this section.

4.3 Alignment of Architecture and Production Schedule

In the previous section, the relationship of AEs and PWAs has been discussed without taking into consideration the project schedule. In this section, we add the production schedule as an additional factor influencing the producibility of a system.

Today, many organizations apply an iterative and incremental development approach. Consequently, during project scheduling, iterations are defined and PWAs are assigned to them. Iterations can have fixed durations, which is called time-boxing, or can have individual durations. By defining iterations and assigning PWAs to them, a basic order is defined

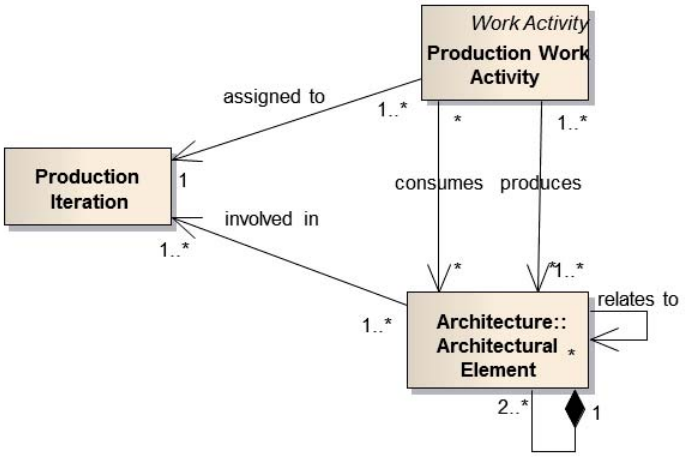


Figure 39: Alignment of Architecture and Production Schedule

on the PWAs. As iterations are performed sequentially, also the PWAs assigned to different iterations are assigned sequentially.

In this section, we assume that a production planner has initially assigned production work activities to production iterations. Hence, relationships of architectural elements to production iterations have implicitly be established.

Figure 39provides an overview on the general relationship of architectural elements, production work activities, and production iterations according to the software production meta-model introduced in Section 3.4. This section covers the excerpt of the software production meta-model shown in Figure 39 with respective metrics.

Similar to the previous section, we introduce respective metrics from an architect's and a production planner's perspective.

4.3.1 Architect's Perspective

The architect is first of all interested in how single architectural elements are related to iterations based on the assignment of production work activities to iterations. This fact can be measured by means of the metrics introduced in the following sub-section.

Architectural elements get involved in several iterations if they are produced by several production work activities that are assigned to different iterations. This is an interesting fact from an architect's point of view, as this means that architectural elements over time undergo several changes.

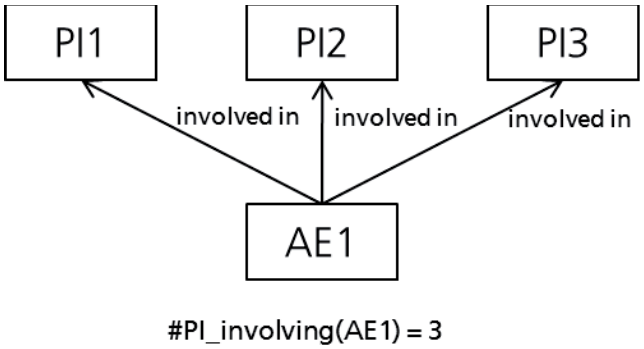


Figure 40: Number of PI involving AE

Metric Definition: *AE_{all}->N: #PI_involving(AE) = n, where n is the number of production iterations involving the architectural element AE, i.e. contributing to the production of the architectural element AE.*

Figure 40 shows an example. The architectural element AE1 is involved in 3 iterations: #PI_involving(AE1) = 3.

It is interesting to consider the values of #PI_involving(AE) and, for instance, #PWA_producing(AE) for a specific architectural element AE in combination. A high value of #PWA_producing(AE) can be an indicator for potential production problems as the AE is modified several times. But if the value of #PI_involving(AE) is low, for instance, one, this means that all the PWAs producing a specific architectural element have been assigned to the same production iteration which could reduce the risk of production problems if, for instance, the resources assigned to the PWAs are co-located and can easily communicate. This example shows that the metrics introduced in this chapter always need to be interpreted in combination in the respective project context.

We assume that iterations are ordered sequentially and also that they are numbered accordingly from 1,…,n.

Consequently, we can add a numbers to each architectural element for the production iteration creating as well as the production iteration finishing it.

Metric Definition: *AE_{all}->N: PI_ creating(AE) = n, where n is the number of the production iteration creating the architectural element AE.*

Metric Definition: *AE_{all}->N: PI_finishing(AE) = n, where n is the number of the production iteration finishing the architectural element AE.*

In the example shown in Figure 40, the architectural element AE1 is created in production iteration PI1 and finished in production iteration PI3.

PI_ creating(AE1) = 1

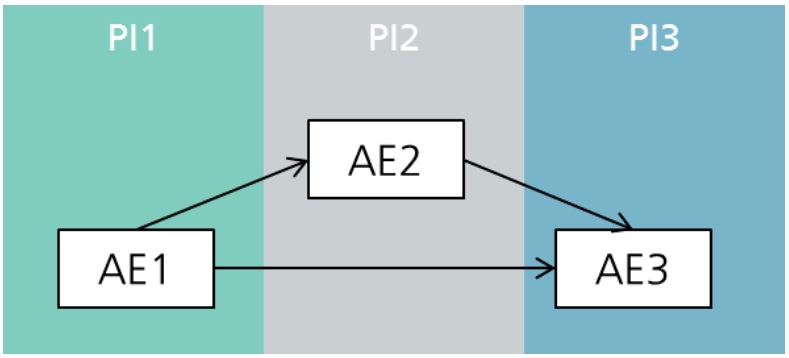


Figure 41: Example - Set of AEs in different PIs

PI_finishing(AE1) = 3

Based on the values of PI_ creating(AE) and PI_finishing(AE) the production duration in number of iterations can be determined.

Metric Definition: *AE_{all}->N: ProductionDuration(AE) = PI_finishing(AE) - PI_ creating(AE) + 1*

In the example shown in Figure 40, ProductionDuration(AE1) = 3.

Again, we also consider the relationship of sets of architectural elements to in this case production iterations. Architectural elements can be related to each other and in combination, for instance, realize a certain behavior. Architectural elements related to each other should ideally be realized in combination to assure that they together fulfill their responsibility. The following metrics help in evaluating situations where related architectural elements are produced in different iterations.

Metric Definition: *$P(AE_{all})$->N: #PI_involving({AE}) = n, where n is the number of iterations involving members of the specified set of architectural elements.*

Metric Definition: *$P(AE_{all})$->N: PI_finishing({AE}) = n, where n is the iteration number when all elements of the specified set of architectural elements are finished.*

Figure 41 illustrates the two metrics. The architectural elements AE1, AE2, and AE3 are related to each other but the production is spread over three production iterations PI1, PI2, and PI3. The set of architectural elements is completely produced after three production iterations. This leads to the following values for the metrics just introduced:

#PI_involving({AE1,AE2,AE3}) = 3

PI_finishing({AE1,AE2,AE3}) = 3

The following section introduces metrics characterizing the alignment of architecture and production schedule from a production planner's perspective.

4.3.2 Production Planner's Perspective

Production planners are interested in how the production iterations they have planned relate to architectural elements.

If we take one single production iteration into account, the main question is how many architectural elements are involved in the respective production iteration. If many architectural elements are involved in a production iteration, this means that a huge part of the overall system is modified in a single production iteration. This can be a problem as it requires much knowledge about the overall system and potentially many experts covering the architectural elements with their knowledge need to be involved. Consequently, the following metric is determined per iteration:

Metric Definition: *PI_{all}->N: #AE_involved_in(PI) = n, where n is the number of architectural elements involved in the production iteration PI.*

In Figure 42, the production iteration PI1 contains three architectural elements, i.e. #AE_involved_in(PI1) = 3.

Another interesting aspect related to production iterations and architectural elements is the coverage of the overall system by a certain production iteration. Each production iteration potentially creates new architectural elements and/or modifies them. In that sense, it is important to know how many architectural elements are involved in a production iteration as well as how this number relates to the overall number of architectural elements making up the system. The following metric is used to express this relationship:

Metric Definition: *PI_{all}-> R_0^+: %System_Coverage(PI) = r, where r is #AE_involved_in(PI) / $|AE_{all}|$.*

In Figure 42, we can determine %System_Coverage(PI1) = 3/7.

As mentioned above, this metric expresses how a production iteration PI covers the architecture of the system. A high value of %System_Coverage(PI) means that a huge portion of the overall system is modified in a production iteration. Consequently, the risk of corrupting the overall system quality is high in such a production iteration. Much knowledge of the overall system is required in the respective production

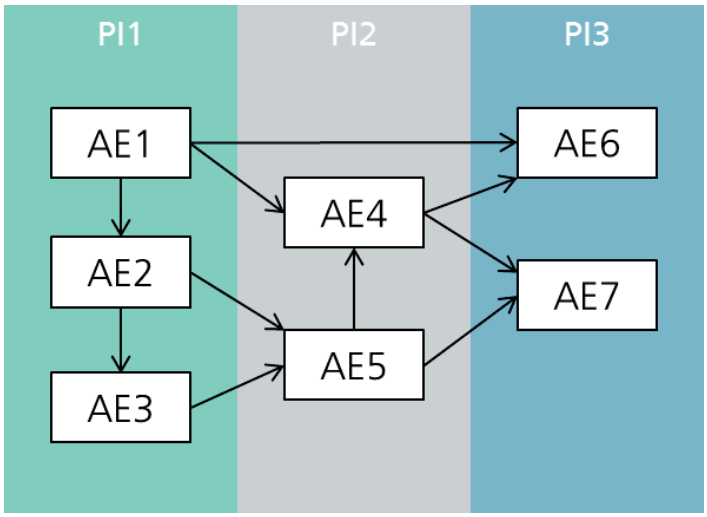


Figure 42: Example - Coupling between Iterations

iteration as many AEs are involved. If the value of %System_Coverage(PI) remains stable within certain boundaries during performing software production, this characterizes a constant growth of the system. In the first production iterations of a project, the value of %System_Coverage(PI) is potentially higher, because many AEs need to be newly created.

Production iterations contribute to producing the system over time. Relating sets of production iterations and architectural elements means to characterize how architectural elements are produced over time and how their relationships influence the iterations that follow each other. If, for instance, production iterations are coupled because the architectural elements involved into them are coupled, or if production iterations share architectural elements as they are modified in both production iterations, this can be a source of potential production problems. The following metrics help in detecting respective situations.

Metric Definition: *PI_{all}->N: Coupling(PI_i) = n, where n is the number of relations from architectural elements produced in PI_i and every architectural element produced in PI_j, $j \neq i$.*

Figure 42 shows an example for production iterations that are coupled. PI1 and PI2, PI2 and PI3, and PI1 and PI3 are coupled.

The respective coupling values are as follows:

Coupling(PI1) = 4

Coupling(PI2) = 6

Coupling(PI3) = 4

Production iterations are typically not only coupled, they also share architectural elements. Two production iterations share an architectural element if they both contribute to the production of the architectural elements. Figure 43 provides an example. The production iterations PI1 and PI2 share the architectural elements AE2, AE3, and AE4. PI2 and PI3 share AE4 and AE5. PI1 and PI3 share AE4.

Sharing architectural elements between production iterations can be an indicator for production problems. On the one hand, it is good that PWAs modifying the same AEs are assigned to different production iterations as otherwise this would be a potential source of conflicts if the PWAs would modify the AEs in parallel in the same iteration. But on the other hand, sharing indicates a certain risk that results of production iteration PI_i get corrupted in production iteration PI_{i+1} if the same areas of the AEs are modified again. This also has an effect on the testing perspective of the project. If a huge number of AEs modified in an iteration are modified again in later iterations, retesting several features again and again is required. Hence, regression testing as a best practice and setting up a regression test suite that can automatically be executed by respective tools should be considered, for example.

The following metrics express if production iterations share architectural elements and characterize the degree of overlapping between the production iterations.

Metric Definition: *PI_{all}->N: #Shared_AE(PI_i): The number of architectural elements shared by PI_i with PI_{i+1} and P_{i-1}. In case i = 1 only the elements shared with PI_{i+1} can be considered.*

Example in Figure 43:

#Shared_AE(PI1) = 3

#Shared_AE(PI2) = 4

#Shared_AE(PI3) = 2

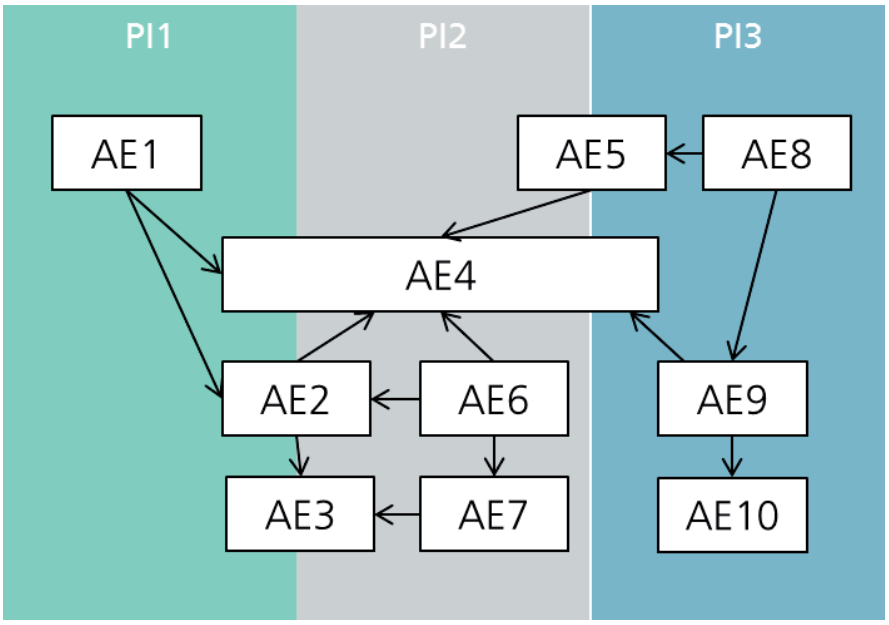


Figure 43: Example - Iterations sharing AEs

Metric Definition: *PI_{all}->R_0^+: %Shared_AE(PI) = r, where r = #Shared_AE(PI) / #AE_involved_in(PI).*

The metric measures the percentage of shared architectural elements for a given production iteration.

Example in Figure 43:

%Shared_AE_PI1 (PI1) = 3/4

%Shared_AE_PI1 (PI2) = 2/3

%Shared_AE_PI1 (PI3) = 2/5

After a certain set of production iterations has been performed, it is important to know how much of the overall system has already been produced. The progress of production can be measured relative to the architectural elements that already have been completely produced, i.e. they are not modified again. This can be expressed by the following metric:

Metric Definition: *PI_{all}->R_0^+: %Completed_AE_after(PI) = r, where r is the percentage of architectural elements that have been completely produced after production iteration PI has been finished.*

This metric can be complemented by another metric that tells us about the number of architectural elements that at least have been initially created after production iteration PI.

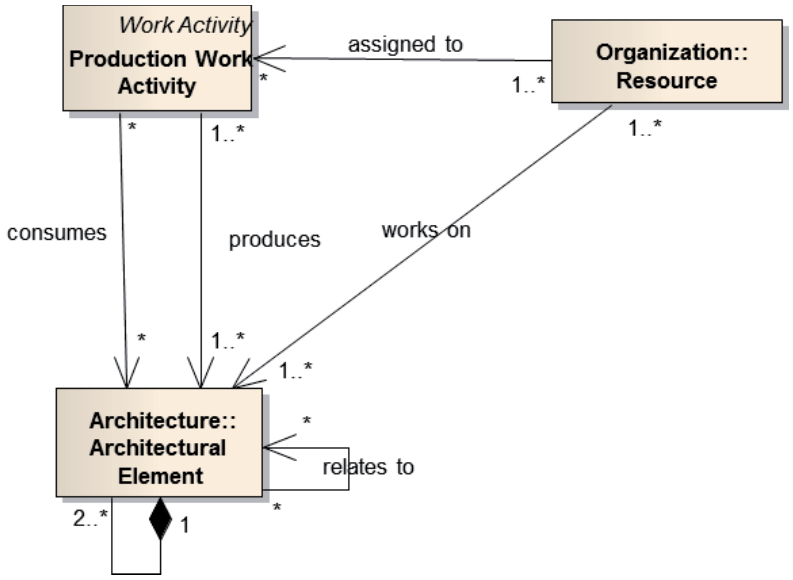


Figure 44: AEs and Resources in Meta-Model

Metric Definition: *PI_{all}->R_0^+: %Created_AE_after(PI) = r, where r is the percentage of architectural elements that have been initially produced after production iteration PI has been finished.*

If an architectural element has been initially produced successfully, this is a first indicator that its production is technically feasible and the risk of production problems related to the architectural element is reduced from that point in time.

4.4 Alignment of Architecture and Resource Assignments

The third dimension of producibility considered in this thesis is the alignment of architecture and resource assignments. During resource assignment, architectural elements are related to teams or single persons both representing resources by assigning respective production work activities to them. Figure 44 shows the excerpt of the software production meta-model that is in the focus of this section.

Similar to the previous sections, we structure this section into the architect's and the production planner's perspective.

4.4.1 Architect's Perspective

Architects are interested in who is taking care of the architectural elements they have designed in their architecture. In general, it is not desirable that many different resources are involved in producing an architectural elements as this requires increased communication and can lead to

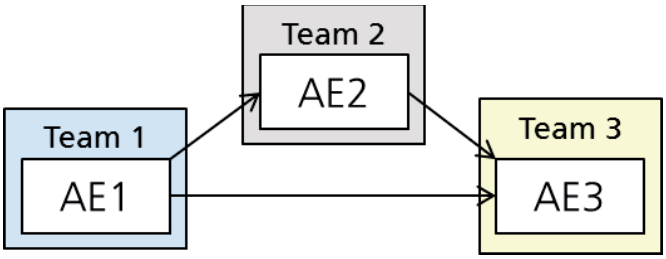


Figure 45: Example - Architectural Elements and Resources

conflicts. Some agile methods, for instance, advocate common code ownership, which allows everyone to perform changes on each architectural element. But the assignment of resources to production work activities in software production limits the resources working on certain architectural elements. It can make sense to assign production work activities involving the same architectural elements to the same resource to reduce the required knowledge transfer related to architectural elements between resources. Resources get the chance to specialize themselves with respect to certain architectural element types.

The following two metrics characterize the relation of single architectural elements and sets of architectural elements to resources. Resources can be either teams or single persons, as already mentioned above.

Metric Definition: *AE_{all}->N: #Resources_working_on(AE) = n, where n is the number of resources working on an architectural element AE during a software production project.*

In an ideal case, only one resource is working on an architectural element to prevent conflicts and required knowledge transfer.

Metric Definition: *$P(AE_{all})$->N: #Resources_working_on({AE}) = n, where n is the number of resources working on a specified set of architectural elements {AE} during a software production project.*

As in the case before, in an ideal case only one resource is working on a set of architectural elements if such architectural elements together are responsible to realize a certain behavior of a system. Typically, this cannot be achieved for the overall system. Certain architectural elements contribute to various behaviors of the system that are realized by different resources.

Figure 45 shows an example. The architectural element AE1 has been assigned to the resource Team1, AE2 to Team3, and AE3 to Team 3. This results in the following values of the metrics introduced above:

#Resources_working_on(AE1) = 1

#Resources_working_on(AE2) = 1

#Resources_working_on(AE3) = 1

#Resources_working_on({AE1,AE2,AE3}) = 3

The following section introduces metrics related to the production planner's perspective.

4.4.2 Production Planner's Perspective

Production planners are interested in the relationship of architectural elements and resources from their specific perspective. They are responsible for resource assignments and need to validate their decisions with respect to the architecture. The following metrics can help in doing so.

Metric Definition: *RES_{all}->N: #AE_worked_on_by(RES) = n, where n is the number of architectural elements the resource RES is working on.*

If a resource is working on many architectural elements, it needs to be capable of doing so regarding skills and availability. A high value of #AE_worked_on_by(RES) can be an indicator of an overloaded resource.

Metric Definition: *Res_{all}->N: Coupling(RES) = n, where n is the coupling of RES with any other RES in Res_{all} determined based on the coupling of the architectural elements produced by RES with any other AE in AE_{all} that is not produced by RES.*

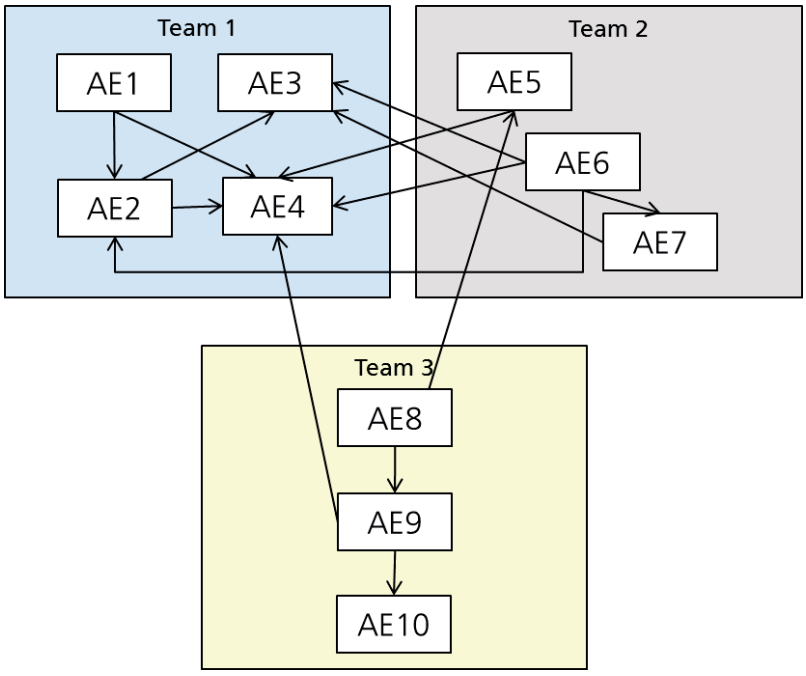


Figure 46: Example - Coupling between Resources

Coupling between resources is illustrated in Figure 46.

The four architectural elements AE1, AE2, AE3, and AE4 have been assigned to Team1, AE5, AE6, and AE7 have been assigned to Team2, and AE8, AE9, and AE10 have been assigned to Team3. Based on the relationships between the architectural elements, relationships between the teams are established. Basically, each team has to communicate with all other teams in this example based on the relationships between architectural elements. If we determine the coupling between teams according to the metric introduced above, we get the following values:

Coupling(Team1) = 2

Coupling(Team2) = 2

Coupling(Team3) = 2

Metric Definition: *RES_{all}->N: #Shared_AE(RES) = n, where n is the number of architectural elements shared by the resources RES with any other RES in RES_{all}.*

Metric Definition: *RES_{all}->R_0^+:%Shared_AE_Resource(RES) = r, where r = #Shared_AE(RES) / #AE_worked_on_by(RES).*

Hence, %Shared_AE_Resource(RES) is the percentage of the number of AEs shared by RES and the overall number of AEs produced by RES.

Sharing an architectural element between two resources means that both resources contribute to the production of the respective architectural element.

Figure 47 illustrates sharing of architectural elements between resources. The architectural elements AE3 and AE4 are shared between Team 1 and Team 2, i.e. both modify them.

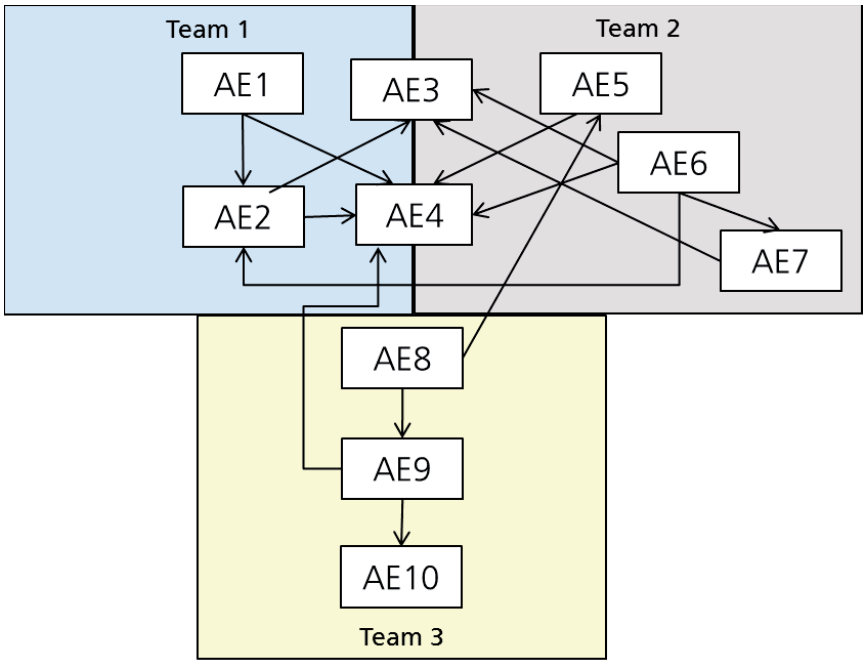


Figure 47: Example - Sharing between Resources

Coupling between resources and sharing of architectural elements between resources means that communication is required between the respective resources. Hence, resource assignments should be performed based on architectural knowledge to prevent communication overhead.

4.5 Context Factors

The metrics presented in Section 4.2, 4.3, and 4.4 can be used to measure the producibility of a system based on a given architecture and production plan. However, the interpretation of the values provided by the metrics needs to be done under consideration of various context factors. The context factors can compensate bad values of certain metrics or even increase the risks related to a bad value. An architectural element AE might be modified often during the course of a project, i.e. for instance, #PI_involving(AE) = 5 while the overall number of production iterations is 6. If the architect is able to explain that AE has an internal structure that facilitates incremental extension of AE, the risks related to multiple modifications of AE is lower than the value #PI_involving(AE) = 5 might indicate.

In this section, context factors that need to be considered while interpreting the producibility metrics introduced before are presented. The context factors are classified into architecture, production process, and organization-related context factors. This means they originate from one of these areas, but can potentially influence all metrics presented before.

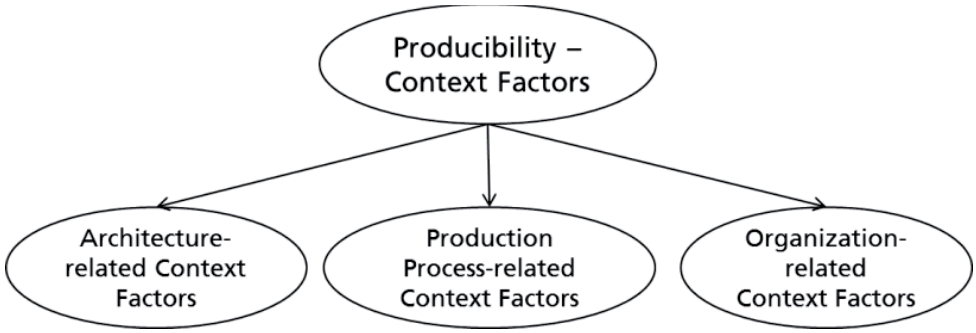


Figure 48: Classes of Producibility Context Factors

Figure 48 shows the three classes of context factors considered in this thesis.

The context factors described in this section are not considered to be complete. In each class of context factors, i.e. architecture related, production process-related, and organization-related, additional factors might be required in a specific project context. The context factors presented in this section are based on our experience in projects in the area of information system development.

4.5.1 Architecture-related Context Factors

This section introduces the context factors related to architecture that should be considered based on our experience. These context factors specifically should be considered if bad values of metrics related to the architect's perspective appear. They can help architects to decide, if potential production problems are related to certain architectural elements and which countermeasures they should take. Figure 49 shows the context factors considered related to architecture.

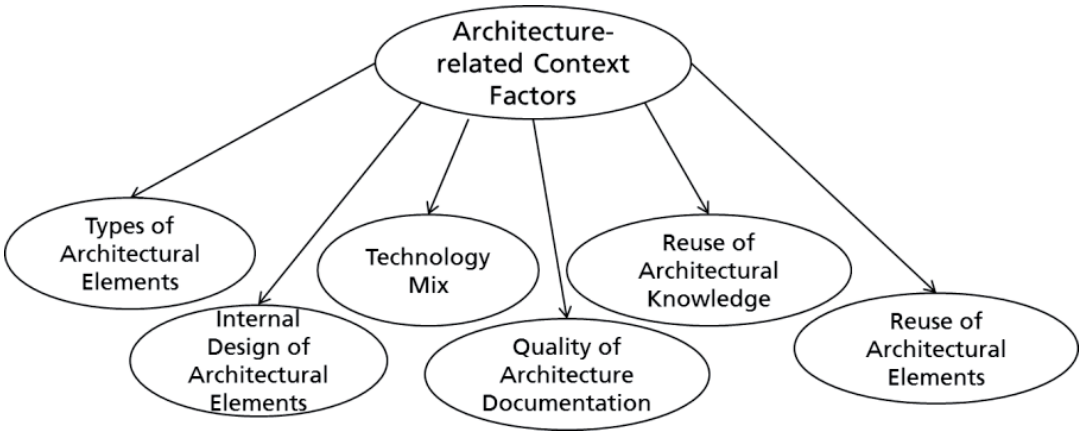


Figure 49: Overview Architecture-related Context Factors

4.5.1.1 Types of Architectural Elements

Architectural elements can be classified into different types. This thesis is based on project experience from the information system domain. One can distinguish infrastructure architectural elements and business architectural elements. Infrastructure architectural elements make up the technical infrastructure where business architectural elements can be embedded. Infrastructure architectural elements are, for instance, workflow engines, enterprise service buses, or rule engines. Business architectural elements are, for instance, services or workflows capturing business specific functionality.

Typically, infrastructure architectural elements are touched more often during a project than business architectural elements. An enterprise service bus, for instance, must be modified several times in a sense that new services must be connected to them. This leads, for instance, to a high value of #PI_involving(Enterprise Service Bus) which might not be a problem here as enterprise service buses are prepared for such kind of changes (maybe only configuration files need to be changed).

4.5.1.2 Internal Design of Architectural Elements

Design constraints and texture

According to Section 2.1, architects are not responsible for designing architectural elements completely. Internal design decisions are left open to designers. But for several reasons, architects can constrain the internal design of architectural elements, i.e. they prescribe the internal structure of an architectural element at least partially. If they put similar constraints to several architectural elements, they define a texture (see Section 2.1).

In software production, constraints put on the internal design of an architectural element can help to reduce certain production risks. Let us consider an architectural element that is changed often during a project. If an architect prescribes an internal structure that makes an architectural element easily extensible, for instance, by a kind of plug-in mechanism, the risk of production problems related to the high number of changes can be reduced. In each production iteration where the architectural element is modified, a separate plug-in could be developed that is largely independent of the rest of the architectural elements, for instance.

Object-oriented Metrics

Object-oriented metrics like complexity or cohesion [Hen95] can be used in addition to characterize the internal design of architectural elements. They can be used to decide if architectural elements are critical for production. If an architectural elements with a high estimated complexity is modified often, more production risks are related to it than in the case of an architectural element with low complexity. High cohesion of an architectural element would be an argument to not split an architectural element, which could be a decision taken during producibility analysis, as

it will be explained in Chapter 5. A high estimated size of an architectural element could be a reason to split it.

4.5.1.3 Technology Mix

Technology decisions heavily influence production and should be thoroughly considered. The resources supposed to perform production must be able to cope with the selected technologies, i.e. they need to be equipped with the required skills. Technologies need to be compatible with the selected tool infrastructure, i.e. appropriate tools need to exist and they need to be integrated into the production environment.

Especially the mix of technologies in a production project is essential for several reasons. A huge number of technologies increases the skills required by the production team, especially if each member must deal with several technologies. Technologies provided by different vendors or organizations might lead to unexpected incompatibilities or at least increased effort to make them work together. When a new web-based information system is planned to be produced, for instance, a general decision to be taken might be to decide to select Microsoft's .NET platform [MSNet11] or JAVA EE as it is supported by Oracle [Java11]. Both provide the required technology required to build a web-based information system. The respective technology platforms come along with comprehensive tool sets. The production team should be used to them to prevent long training periods in the project. The combination of .NET and JAVA EE solutions can prevent the production team to use the respective technologies as planned by the respective vendors.

An architect should evaluate the mix of technologies carefully with respect to production and plan together with the production planner when the respective technology enters the project. If the first production iteration, for instance, is bothered by the introduction of several technologies new to the production team, this can lead to delays, quality issues, etc. in the beginning of the project.

4.5.1.4 Quality of architecture documentation

The architecture documentation is a main point of reference for the production team. The architecture specifies the architectural elements they are supposed to produce. It should contain all architectural information relevant to start producing the architectural element including interfaces, design constraints, selected implementation technologies, etc.

View-based architecture documentation

The meta-model of software architecture introduced in Section 2.1 refers to best practices in architecture documentation. Architecture documentation should contain different architectural views as well as descriptions of the main architectural decisions. From a production point of view, it is specifically important that the architecture documentation contains for each architectural element including interfaces, design constraints, selected implementation technologies, as already mentioned above. Especially, the elements that are supposed to be critical for any reason, for instance, because they are modified often or need to be finished early, the architecture documentation must contain such information, otherwise production problems can be the consequence.

Design documentation

Besides architecture documentation, documentation of designs for each architectural element is important. Hence, production risks can be reduced if for each (critical) architectural element design documentation is available, for instance, according to the KoBrA approach [ABB+01]. In the KoBrA approach, each architectural element is documented by means of different models, for instance, structural model, behavioral model, or functional model.

4.5.1.5 Reuse of Architectural Knowledge

The maturity of software architecture as a Software Engineering discipline becomes visible in many handbooks of software architecture [BCK03] [TMD10] [RH08]. Similar to other engineering disciplines, such handbooks capture the existing knowledge on software architecture.

Architectural knowledge can be captured in various ways, beyond the most prominent ones being architectural styles, tactics, patterns, reference architectures, or product line architectures. As producibility is not yet explicitly addressed in software architecture research, architecture handbooks do not capture knowledge on producibility of architecture systematically. Nevertheless, they should be used as a reference by architects also from a producibility point of view. Using architectural styles, tactics, patterns, etc. from the existing software architecture literature can help to reduce production problems. The reuse of proven solutions in general helps not to repeat mistakes done in the past again and reduce production problems. Furthermore, a common vocabulary between architects but also between architects and the production team is established that helps to ease communication, prevent misunderstandings and resulting production problems.

Architects should check with respect to producibility, if they can refer to existing architectural solutions instead of inventing everything from scratch or using existing solutions under different names. In the software production meta-model, for instance, an optional relationship of an architectural element to an architectural element type is modeled. In an ideal case, each architectural element is of a known type, i.e. of a type

that is referenced somewhere in the organizations or overall architecture body of knowledge.

4.5.1.6 Reuse of Architectural Elements

Besides reuse of architectural knowledge, reuse of concrete architectural elements should be considered with respect to software production and producibility. In many cases, reuse approaches like product line engineering or component-based development have proven to be successful in reducing time to market and effort or increasing quality [HOF11]. Hence, from a producibility perspective, it should be analyzed how an architecture makes use of already existing architectural elements. These can be code libraries, executable components, or specific frameworks that are available in an organization, from other vendors, or open source.

Building architectural elements based on reuse can have positive as well as negative effects on the producibility of a system. If an architectural element has been used before and can be reused without modifications, for instance, the risk of production problems is lower than in the case of developing an architectural elements newly from scratch. But if an architectural elements is supposed to be reused but has to undergo many modifications during the project this can increase the risk of production problems. The documented knowledge on the reused architectural element might be limited and its reconstruction might cause additional effort and time.

In the case of each architectural element, reuse needs to be considered carefully from a producibility perspective.

In the following section, context factors related to the production process are introduced.

4.5.2 Production Process-related Context Factors

This section introduces the context factors related to the selected production process that should be considered based on our experience. These context factors specifically should be considered if bad values of metrics related to the production planner's perspective appear. Figure 50 shows the context factors considered related to the production process.

4.5.2.1 Support by Production Work Activity Types

As introduced in Chapter 3, production is essentially product-oriented and a production process provides product-specific guidance. Production work activity types guide the production team in producing certain types of architectural elements (s59ee Section 3.5.2 for an example).

From a producibility perspective, it is important that the majority of architectural element types that are used in a system are supported by production work activity types, i.e. the optional relationship between architectural element types and production work activity types in the production meta-model exists in a specific instance. Especially architectural elements or element types that are supposed to be critical because of some other producibility metric should be supported by a defined production work activity type. If workflows or services, for instance, are supposed to be built many times as part of a software production project, a detailed process how to create a workflow or service, i.e., production work activity types for workflows and services, are required.

Production planners should check, how well the production work activities of a software production project are covered by production work activity types. The higher the coverage especially of architectural elements with bad values for certain producibility metrics, the more likely certain production problems still can be prevented.

4.5.2.2 Support by Development Activity Types

Production work activity types involve development activity types as defined in the software production meta-model (see Section 3.4). Development activity types refer, for instance, to design, implementation, or

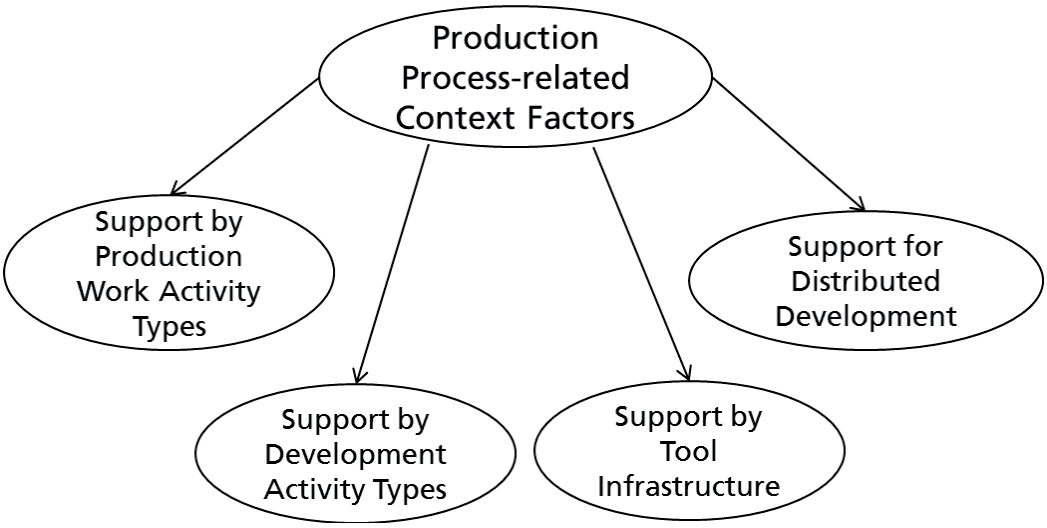


Figure 50: Overview Production Process-related Context Factors

testing best practices to adopted in the context of a production work activity.

Development activity types can provide specific support for software production and should be considered from a producibility perspective. Based on our experience, production problems can be caused, for instance, by missing architecture compliance. Reuse can be increased by higher architecture compliance [KMH+08]. During production, architecture violations are introduced by, for instance, introducing unplanned dependencies between architectural elements, that can cause unexpected problems in later production iterations as side effects can occur. If compliance checking is foreseen as a development activity type to be performed in certain production work activity types, architecture violations can be detected early before they cause further production problems. Compliance checking can be performed during production, for instance, by means of the approach proposed in [Kno09].

It should be considered by production planners, which development activity types specifically support production.

4.5.2.3 Support by Tool Infrastructure

Tools have a high potential to increase the producibility of a system. They can support production work activities or even completely automate them. Software production does not aim at maximizing the automation of the production process. Automation requires investment and not each production work activity or development activity type is a good candidate for automation, as the initial investment potentially never pays off. Approaches like Software Factories [GSC+04] or software production lines [Kru06] heavily rely on tool support. McGregor provides guidelines on how to set-up production environments for software product lines [McG05] as one software production scenario (see Section 3.3).

Production work activity types that are recurring often in a project are good candidates for tool support. They could be supported by a tool infrastructure that is aware of the process [Rom03]. In general, production planners should evaluate especially for each critical production work activity in a project, for instance, because it has a high value of a related producibility metric (see Section 4.2.2), if appropriate tool support is available or can be set-up easily.

4.5.2.4 Support for Distributed Development

More and more, software is produced in a distributed fashion. Internal and external units (see Section 2.2.5) can be involved in a software production project. Hence, software organizations set-up supply chains similar to other engineering disciplines. Specialized suppliers take over pro-

duction work activities and deliver architectural elements to be integrated into the overall system.

Distributed development should be well supported to prevent related production problems. Communication between distributed teams, for instance, needs to be enabled by means of adequate processes and tools. It is important, for instance, to agree on a specific format how production work activity specifications are exchanged with suppliers. Bug and issue tracking tools can be used to report on production problems. Common repositories should be used to exchange the results of production work activities. Furthermore, task assignment in distributed settings can be supported by empirical models [LM10].

For each external unit respectively for each set of teams that are supposed to work together but are geographically distributed it should be checked, if adequate support for collaboration is available. Otherwise, there is a certain risk of production problems because of communication problems. Such communication problems can be further intensified because of cultural differences.

4.5.3 Organization-related Context-Factors

Finally, the organizational context needs to be considered while analyzing producibility based on the producibility metrics introduced before. Figure 51 shows the context factors that are considered in this section.

4.5.3.1 Capabilities of Resources

Single resources or teams must be capable of performing the production work activities that are assigned to them. Several skills are desirable. If the production work activity relates to a certain production work activity type or architectural element types, the resource should be familiar with the respective types. If a resource is supposed to build a service, for in-

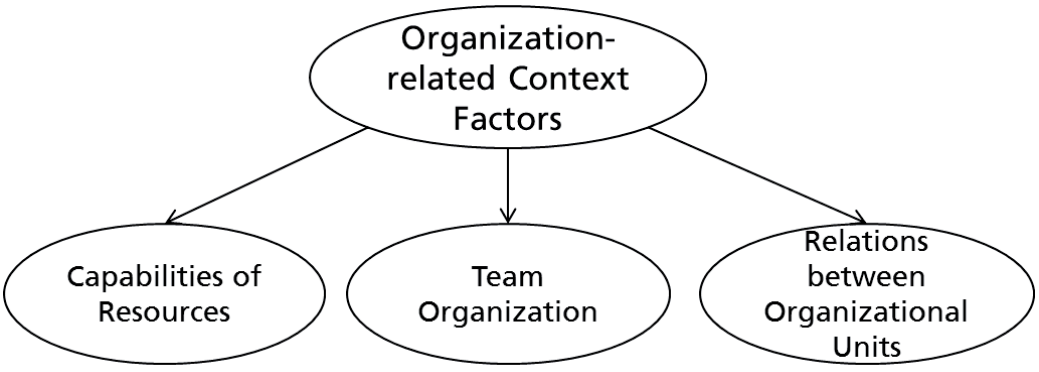


Figure 51: Overview Organization-related Context Factors

stance, in an ideal case, the resource has performed similar production work activities before.

Familiarity with the selected technologies is another important context factor related to the assigned resources. Production planners and architects should consider selecting different technologies if they can be easily substituted comparably and the skill profile of the resources fit better afterwards.

4.5.3.2 Team organization

Project (sub-) teams need to be organized in a certain way. Section 2.2.5 introduced different models of team organization, for instance, the chief programmer model or structured open teams.

The team organization can influence producibility in different ways. In the case of architectural elements that are modified often it might be helpful to have a chief programmer as a coordinator. The same holds true for complex architectural elements where a chief programmer could bring in his or her experience to produce the key parts.

In the case of different teams with close coupling or even shared architectural elements (see Section 4.4), it also might be helpful to have chief programmers that can perform the major part of the required communication between teams to canalize the flow of information.

Production planners should check the team organization and relating communication structures between teams to identify potential production problems.

4.5.3.3 Relations between Organizational Units

Production project teams often have to interact with other organizational units. In the case where a project team uses, for instance, a certain framework that is developed in another organizational unit, a dependency between the respective units is established. The project team has to consider, for instance, that the framework team might not be able to answer requests quickly as they have to cope with many requests of various project teams. This can result in production problems if delays occur.

It could also be the case that certain development activities as part of production work activities are performed in different organizational units. Testing is a prominent example that often is performed in a separate test department to assure independence of implementers and testers. The coordination of production work activities with a test team is important to prevent production problems up-front.

Production planners should consider the relationships of the project team to other organizational units and check, if the collaboration is well supported by the production process.

As already mentioned, context factors are important to be considered while interpreting the producibility metrics introduced above. The values of the context factors can compensate significant values of producibility metrics or even make the situation worse.

As we will describe in Chapter 5 in more detail, expert judgment is required to finally decide if a significant value of a producibility metric will lead to production problems in a given context.

4.6 Related Work

This section provides an overview on related work regarding the quality property producibility.

Producibility in the literature

The term producibility of software systems has been mentioned in the literature before. In [Cam07], producibility is defined as follows:

Producibility is "the ability to deliver needed capability in a timely, cost effective, and predictable manner."

Another definition of producibility is given in [NRC10]:

Software producibility is "the capacity to design, produce, assure, and evolve software-intensive systems in a predictable manner while effectively managing risk, cost, schedule, quality, and complexity."

Such definitions of producibility have a broader scope than the one given in this thesis. They refer to the general capability of an organization to produce software, i.e. it refers to the general software engineering capabilities of an organization. In [NRC10], for instance, besides project planning/management or software architecture various other disciplines of software engineering like requirements engineering or process management are mentioned and their role for the overall production capability of an organization (in that case of the US Department of Defense (DoD)) is discussed. While not underestimating the importance of other disciplines of software engineering for the overall production capability of an organization, this thesis uses the term producibility in a closer sense to narrow the scope and be able to make an in-depth contribution related to the alignment of architecture and project plans.

Buildability as related quality attribute

A quality attribute related to producibility as defined in this thesis is buildability. In [BCK03], buildability is introduced as a property of a software architecture. Buildability is defined as "the ease of constructing a desired system based on the architecture by the available team in a timely manner". Hence, buildability establishes a general link between architecture and project planning aspects like time to market. The system, i.e., in this case the architecture, should also be open to certain changes as development progresses. The decomposition of the system into modules, the assignment of modules to development teams and limiting the dependencies between modules, i.e., reducing coupling, are mentioned as important influencing factors of buildability. But more detailed guidelines to decide if an architectural element is critical with respect to production are not given.

In this thesis using the term producibility is preferred over using buildability for several reasons:

- Producibility per definition establishes a link between the architecture and a project plan and addresses the misalignment of architecture and project plan as motivated in Chapter 1.
- Producibility refers to the notion of software production as it has been introduced before, i.e. producibility assumes the adoption of the ideas of software production.
- Producibility suggests a link to other disciplines like manufacturing where the term production is more common and refers to a process that is optimized to be run efficiently (eventually many times) to come up with goods fulfilling their requirements. This thesis wants to emphasize the importance of optimizing the alignment between production plans and the design similar to other engineering disciplines.
- Buildability can be misinterpreted as referring only to the technical build process typically conducted as part of software production but not to the whole production process.

Related Metrics

Object-oriented metrics like coupling [Hen95] [SMC74] inspired the definition of the producibility metrics introduced in this chapter. In general, coupling between architectural elements, for instance, can be used as an indicator for producibility. High coupling between architectural elements, for instance, bears the risk that teams assigned to such architectural elements cannot independently realize them or that dependencies to earlier production iterations cause production problems.

However, the producibility metrics defined in this chapter based on the idea of coupling like, for instance, Coupling(PWA), Coupling(PI), or Coupling(Res) are more precise in measuring the producibility of a system. First, they take into account architectural elements and project planning elements. Furthermore, they filter out values of coupling between architectural elements that are not necessarily relevant for the overall producibility. If, for instance, architectural elements that are highly coupled are

assigned to the same team, the value of Coupling(Res) is not increased, as the team can often deal with the "internal" coupling of architectural elements. By relying on values for coupling between architectural elements to measure producibility, "internal" coupling would be indicated as critical and potential production problems would be pointed out.

In the following chapter, the producibility analysis method is described in detail.

5 Producibility Analysis Method

This chapter introduces the producibility analysis method. The chapter is structured as follows. First, an overview of the producibility analysis method is provided. In the following, each phase of the method is introduced and discussed in detail and an overview on the available tool support is provided.

5.1 Method Overview

Goals of the method

The goal of the producibility analysis method is to identify critical architectural elements, critical project planning elements that are supposed to cause production problems like delays or effort overhead. The concrete outputs of the producibility analysis method are the identified critical elements and recommendations on how to prevent production problems eventually caused by such critical elements. In that sense, the producibility analysis method addresses the practical problem introduced in Section 1.1 as it aims at reducing the misalignment of software architecture and project plans.

The foundation of the producibility analysis method is the integrated meta-model of software production presented in Section 3.4.3 and the quality model of producibility presented in Chapter 4.

Scientific Contributions of the method

The producibility analysis method addresses the practical problem by contributing to solve the scientific problems of missing enforcement of communication between architects and project or production planners (SP1) and missing support for the identification and analysis of critical architectural and project or production planning elements (SP2) discussed in Section 1.3. Thereby, it adopts the idea of a manufacturability analysis known from other engineering disciplines.

Prerequisites of the method

As shown in Figure 52, the producibility analysis method requires as an input the current version of the planned architecture of the product to be produced and the current version of the production plan of the product. The following assumptions are underlying such input products:

- The architecture at least must contain a structural view of the system, i.e., for instance, a Module View or a Component and Connector View.
- The production plan at least must contain a sequence of planned production iterations, an assignment of production work activities to

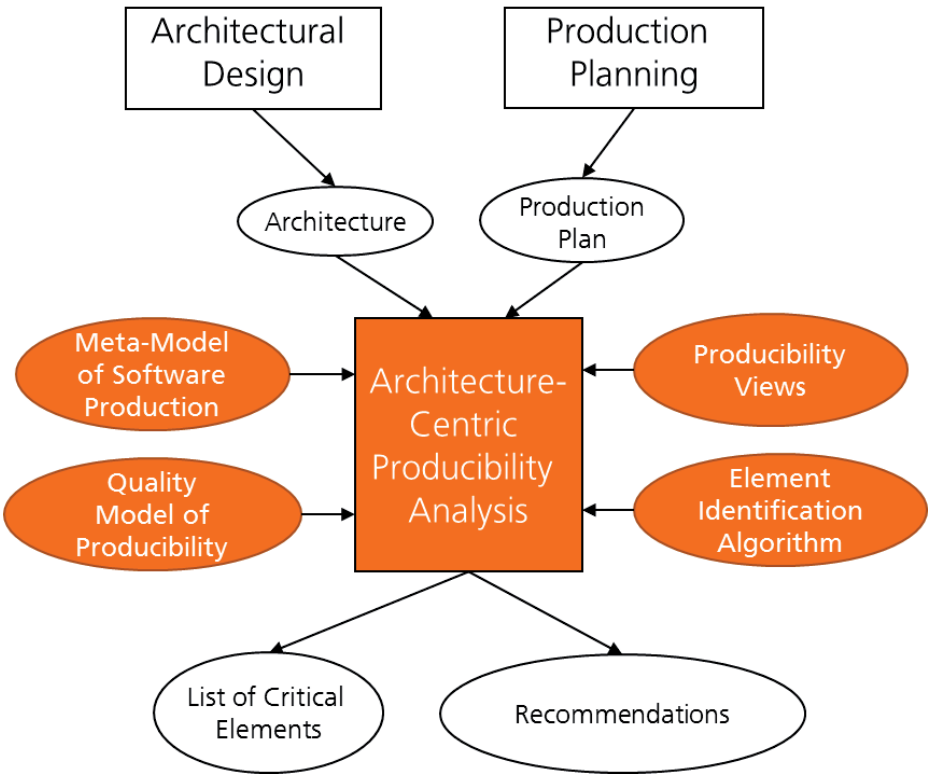


Figure 52: Overview Producibility Analysis Method

production iterations, and an assignment of resources to such production work activities.

– The level of granularity of architecture and production plan used for the producibility analysis must be aligned to each other. If, for instance, the architecture is specified on a level of layers, clusters, and subsystems, the production work activities defined in the production plan must be on the level of layers, clusters, and subsystems or functional domains covered by such architectural elements, too. Production work activities specified on the level of single features require a more detailed view of the architecture to enable a solid producibility analysis.
– Additional architectural views are required as input to the producibility analysis, if specific producibility scenarios are supposed to be analyzed. If, for instance, the architect wants to assure, that a certain behavior of the system is available at a certain point in time, such behavior and the related architectural elements need to be specified in a behavioral view of the architecture.

Producibility scenarios have already been mentioned before. They describe requirements into production from the point of view of different stakeholders like architects, production planners, customers, product managers, etc. and are an optional input to the producibility analysis method. In contrast to a first version of the architecture and the production plan as specified above, we do not expect that the producibility sce-

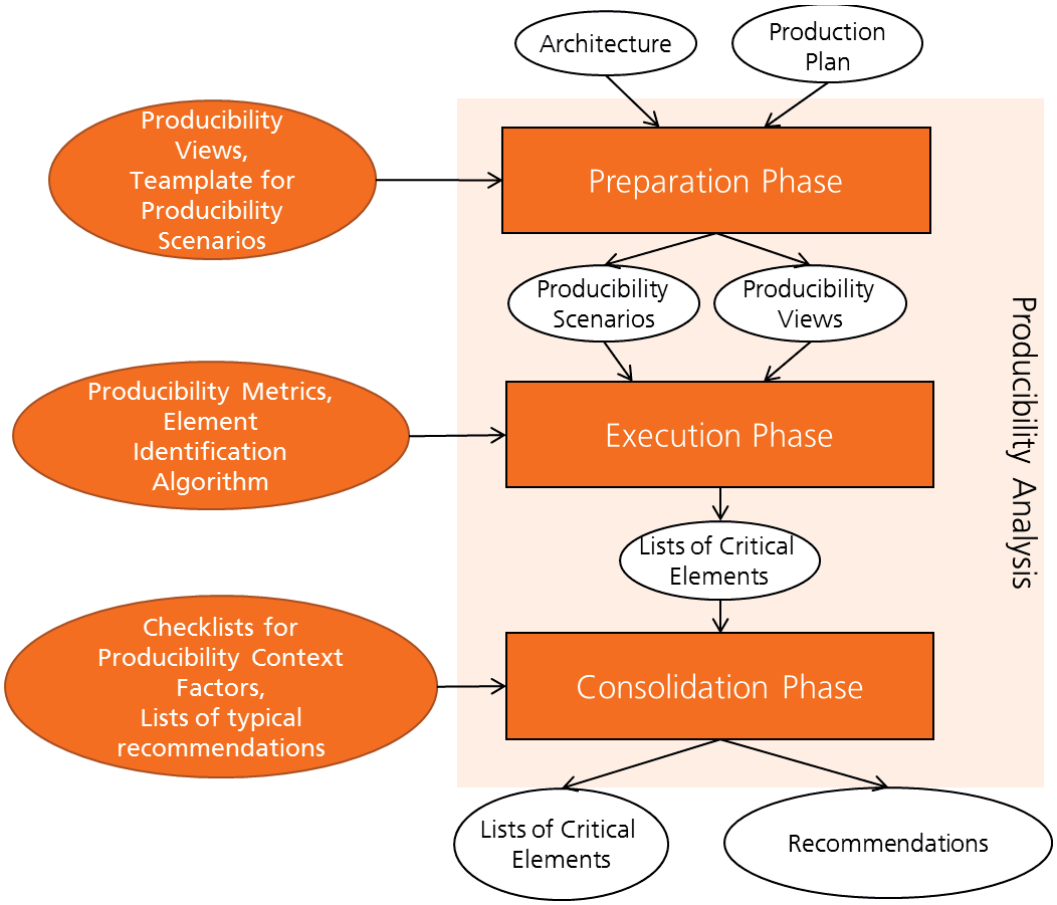


Figure 53: Phases of the Producibility Analysis Method

narios are available from the beginning of a producibility analysis. Rather, we elicit the producibility scenarios in the preparation phase of the producibility analysis if required.

The producibility analysis method consists of the three phases preparation, execution, and consolidation (see Figure 53).

Preparation Phase

The preparation phase creates the inputs for the execution phase. Producibility scenarios are elicited as a first step. The method provides guidance in this step by providing lists of typical producibility-related concerns of different stakeholders that then can be used in interviews with such stakeholders to elicit producibility scenarios. Furthermore, the preparation phase comes up with producibility views that are the basis for determining the producibility metrics introduced in Chapter 4 and for the overall producibility analysis. Three producibility views relating to the three dimensions of producibility are used in this thesis. The producibility views can be modeled by means of the tool Enterprise Architect (EA) [EA11], that has been extended with the capability to model producibility views. The producibility scenarios and the producibility views for a specific production project are the major outcome of the preparation phase of the producibility analysis method.

Execution Phase

The execution phase takes the input of the preparation phase and determines critical architectural and production planning elements based on the producibility metrics introduced in Chapter 4. Architectural elements and production planning elements are supposed to be critical as soon as the respective producibility metrics exceed a certain threshold. All producibility metrics referring to single architectural or production planning elements can be determined automatically based on the producibility views.

Consolidation Phase

In the consolidation phase, the results of the execution phase are analyzed and recommendations are derived. Architects and production planners use checklists that are based on the producibility context factors to decide if an architectural or production planning element marked as critical in the execution phase is really supposed to be critical in the given context. Based on the consolidated list of critical elements they derive recommendations on how to prevent the occurrence production problems caused by the critical elements. Recommendations can be, for instance, to split an architectural element, to change the assignment of production work activities to production iterations, or to assign production work activities differently to the available resources.

In the following section, the different phases of the producibility analysis method are discussed in more detail.

5.2 Preparation Phase

In the preparation phase of the producibility analysis method, the input required for the execution phase is prepared. The preparation phase is made up of several steps as shown in Figure 54. Producibility scenarios are elicited and the producibility views are modeled. Such two steps can be performed in parallel. After producibility scenarios are elicited and documented and producibility views are modeled, the producibility scenarios can be mapped to the producibility views. Architectural and production planning elements modeled in the producibility views and involved in a producibility scenario are added to the documentation of the respective producibility scenario.

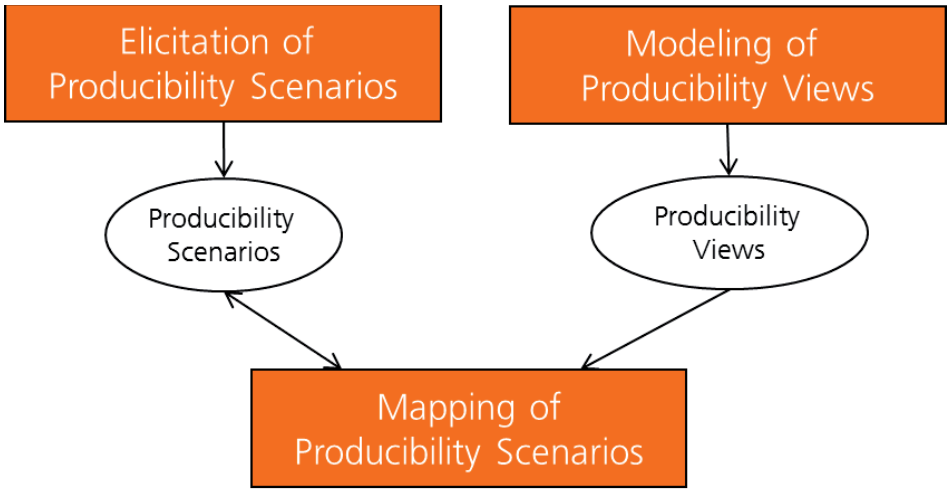


Figure 54: Steps of the Preparation Phase

The concrete outputs of the preparation phase are:

- Producibility scenarios including mapped architectural and production planning elements
- Producibility views

In the following sub-sections, each step of the preparation phase is explained in detail.

5.2.1 Elicitation of Producibility Scenarios

5.2.1.1 Producibility Stakeholders and their Concerns

Producibility Stakeholders

Various stakeholders with specific concerns regarding producibility exist. Architects, production planners, and customers are the main stakeholders. Architects and production planners are concerned with software production by definition. Customers (or also product managers) are important stakeholders as they typically request certain features to be delivered in specific releases. Hence, customers or product managers heavily influence the production schedule.

As further stakeholders, the production team executing the production work activities and its concerns like familiarity with the selected technologies or appropriate tool support should be considered. Operators require adherence to their deployment rules and cycles. Certification units require receiving releases long enough before deployment by the operators is planned, etc.

Producibility Concerns

Each stakeholder has certain typical concerns regarding producibility. For two of the main stakeholders, namely architects and production planners, Table 2shows examples of typical concerns. As we can see, architects mainly aim at early feedback on their architectural decisions. With such typical concerns in mind, architects, production planners can be interviewed and producibility scenarios can be elicited.

Typical Concerns Architect	Typical Concerns Production Planner
Early fulfillment of quality requirements	Project stays within time and budget
Early feedback on the appropriateness of technologies	Release plan can be met
Early feedback on reuse decisions	Teams can work independent of each other
Early feedback that integration works as planned	Suppliers are integrated properly
Early feedback that developers are able to adhere to the architecture	Project team is used to capacity
…	…

Table 2: Examples of Producibility Concerns

The next section introduces producibility scenarios in more detail.

5.2.1.2 Producibility Scenarios

In general, producibility scenarios precisely describe requirements of different stakeholders related to the quality aspect producibility. In that sense, the idea behind producibility scenarios is similar to architectural scenarios that are used to precisely describe quality requirements in the architecture [CKK01]. The main difference is that they do not only affect the architecture but also the production plan.

Producibility scenarios explicitly or implicitly draw links between instances of elements of the architecture meta-model and instances of elements of the production planning meta-model. They specify, for instance, how architectural elements are supposed to be related to production iterations or to resources in a concrete case. This would be an explicit relationship between architectural elements and production planning elements. Instead of directly referring to architectural elements, producibility scenarios can, for instance, also refer to certain features or feature groups. In this case, the relationship of architectural elements and production planning elements is specified implicitly. The following two examples illustrate this:

Example Producibility Scenario

Producibility Scenario 1: All architectural elements contributing to system monitoring must be delivered within the first release to enable continuous checks of system health (by the operator).

Producibility Scenario 2: Three (aggregated) architectural elements that are loosely coupled should be foreseen in the architecture to ease the assignment to the three teams and reduce communication overhead.

Producibility Scenario 3: The functionality to support applying for business trips and supporting accounting of business trips must be completely available after two releases.

Producibility Scenario 1 and 2 explicitly refer to architectural elements. Producibility Scenario 3 implicitly refers to architectural elements. The functionality to support applying for business trips is realized in certain architectural elements but the mapping of the functionality to architectural elements needs to be figured out before the producibility scenario can be analyzed. Section 5.2.3 refers to this mapping.

Furthermore, the examples show that a producibility scenario can be driven rather from the architect's, the production planner's, or the customer's perspective. Producibility Scenario 1 will rather be driven by an architect because one of the architect's producibility concerns is to assure the fulfillment of certain quality requirements early. Producibility Scenario 2 will rather be mentioned by a production planner that wants to use the available resources most effectively. Producibility Scenario 3 first of all addresses a concern of a customer.

Producibility Scenario Template

Producibility scenarios can be documented according to the template shown in Table 3.

Name	Name of the scenario
Stakeholder	The main stakeholder interested in this scenario.
Producibility Dimension	Dimension selected from the quality model of producibility (alignment of architecture and production WBS, alignment of architecture and production schedule, alignment of architecture and resource assignments)
Description	Description of the scenario
Involved AE	The instances of architectural elements referenced in the scenario
Involved PPE	The instances of production planning elements (e.g. production work activities, production iterations, resources) referenced in the scenario

Table 3: Producibility Scenario Template

As already discussed before, a producibility scenario is always based on a concern of a specific stakeholder. Hence, the stakeholder is part of the producibility scenario template.

Each producibility scenario is related to one of the producibility dimensions of the quality model of producibility. This gives each producibility scenario a clear focus and does not mix up different aspects of producibility.

The core of the producibility scenario is the description. Here, the relationship of instances of elements of the architecture meta-model and instances of elements of the production planning meta-model is described as precise as possible. After that, the involved architectural and production planning elements are made explicit in the rows "Involved AE" and "Involved PPE". If the involved architectural and production planning elements are not yet precisely known they can be added in the mapping step of producibility scenarios (Section 5.2.3).

5.2.1.3 Elicitation Process

The elicitation process for producibility scenarios consists of three major steps. First of all, producibility stakeholders are identified. Then, interviews with the identified stakeholders are conducted. Finally, the producibility scenarios are documented.

Identification of Stakeholders

The first step of the elicitation process is the identification of producibility stakeholders. Concrete persons that are good candidates to represent the stakeholder role need to be selected in an organization, for instance, by the production planner.

Interviewing Stakeholders

After stakeholders have been identified, they are interviewed. Interviews with stakeholders are the main source of information to formulate producibility scenarios. Requirements documents, for instance, can also be a source of information, but as producibility is not systematically considered in software projects today, we do not rely on information inside the existing documentation.

Communication between architects and production planners can already be established in this initial step of the producibility analysis method. Architects and production planners as typical producibility stakeholders can interview each other to elicit producibility scenarios, for instance.

Documentation of Producibility Scenarios

During and after performing the interviews producibility scenarios are documented. The template introduced in Section 5.2.1.2 can be used in this case.

While in the description part of the scenario, for instance, architectural elements or production planning elements might be summarized in for-

mulations like "all architectural elements contributing to system monitoring" or "three teams" and not listing them one by one, this is done in the rows "Involved AE" and "Involved PPE" in the producibility scenario template. These two rows are essential for modeling the producibility scenario and determining producibility metrics. In first versions of a producibility scenario documentation, involved AE and involved PPE do not necessarily need to be listed. This might disturb the process of initial elicitation and documentation of producibility scenarios as it requires additional effort to be spent by the architect to map the description part of the scenario precisely to architectural elements and production planning elements. But later on in the step "Mapping of Producibility Scenarios" (Section 5.2.3) they need to be added based on the architecture documentation or by interviewing architects and production planners.

5.2.2 Modeling of Producibility Views

The determination of producibility metrics in the execution phase of a producibility analysis is based on producibility views. Such producibility views are used to specify the relationships of architectural elements and production planning elements. Each producibility view focuses on one dimension of producibility and can be used to determine a specific set of producibility metrics.

Three producibility views are used in this thesis:

- Production Work Activity View
- Production Iteration View
- Resource Assignment View

Each of the views shows the relationship of architectural elements and the production planning element referenced in the name of the producibility view, i.e. production work activities, production iterations, and resource assignments. Hence, the three views also relate to the three dimensions of producibility introduced in Chapter 4. In principle, the three producibility views have already been introduced in Chapter 4. The visualizations of examples of the producibility metrics use the ideas behind the producibility views. Figure 37 shows a production work activity view, Figure 43 a production iteration view, and Figure 47 a resource assignment view.

The modeling of the producibility views starts with the production work activity view. Afterwards, the production iteration view and the resource assignment view can be modeled.

Modeling the Production Work Activity View

As a prerequisite for conducting a producibility analysis, the production plan must contain a production work breakdown structure (see Section 5.1) and a structural view of the architecture. Based on these inputs, the production work activity view can be modeled.

If we assume production work activities referring to elements of the requirements like features (see Section 2.2.2), we need to map such a feature-based specification of work onto architectural elements as a first step. In this case, traceability is essential.

Traceability

Traceability from the requirements to architectural elements is required to facilitate this initial mapping step. In an ideal case, a traceability matrix similar to the one shown in Figure 2 allows the identification of architectural elements based on features that are referenced in production work activities. If such traceability information is not made explicit in any form, it can be an effort-intensive task to derive the mapping between features and architectural elements based on, for instance, interview with architects and requirements engineers. If we assume a software project that is performed based on state-of-the-art Software Engineering methods, we can assume that the traceability information is made explicit. Unfortunately, the state-of-the-practice is that in many cases such traceability information at least partially needs to be elicited.

Based on the mapping of production work activities and architectural elements the production wok activity view can be documented. The modeling of production work activity views is supported by extensions made to the tool Enterprise Architect. An overview on available tool support is given in Section 5.4.2.

Modeling of the Production Iteration View

The production iteration view can be modeled based on the production work activity view and the production schedule. The production work activity view provides the information on which architectural elements are related to production work activities. The production schedule contains the information on the assignment of production work activities to production iteration. Consequently, the relationships of architectural elements and production iterations can be derived and modeled.

Modeling of the Resource Assignment View

Besides the production iteration view, the resource assignment view can be modeled after the production work activity view is available. The assignment of resources to production work activities is part of the production plan. The mapping of production work activities and architectural elements is known from the production work activity view. Consequently, the relations of resources and architectural elements can be modeled.

5.2.3 Mapping of Producibility Scenarios

Producibility scenarios describe desired relationships between architectural elements and production planning elements, as mentioned in Section 5.2.1.2. However, the architectural and production planning elements sometimes are not referenced explicitly. In the example *Producibility Scenario 1*, for instance, all architectural elements contributing to system monitoring and the first release are referenced. The architectural elements contributing to system monitoring and the iterations making up the first release need to be identified to analyze the fulfillment of the producibility scenario. Hence, the producibility scenarios need to be mapped to architectural and production planning elements to enable their analysis.

Each producibility scenario is potentially related to a set of architectural elements and a set of production planning elements. Such elements are added to the documentation of producibility scenarios according to the template presented in Section 5.2.1.2. These sets are input to the producibility metrics related to sets of architectural elements and production planning elements determined in the execution phase.

If the general description of a production scenario cannot be mapped to architectural elements and production planning elements, this is an indicator that the architectural design and/or the production plan have not yet been worked out in enough detail. In an ideal case, based on the description part of a producibility scenario it should be possible to identify all relevant elements based on the documentation of the architecture, the production plan. If this is not possible, at least the architect and the production planner should be able to list the respective elements verbally. Unfortunately, based on our experience in many cases the documentation is not sufficient and architects and production planners need to be interviewed to identify the respective elements. The architecture documentation and the production plan should be extended with additional producibility related information if required.

5.3 Execution Phase

In the execution phase, potentially critical architectural and production planning elements are identified. The producibility views modeled in the preparation phase serve as input. All producibility metrics introduced in Chapter 4 can be determined based on the three producibility views. Therefore, the producibility views are processed by an analysis algorithm that detects critical architectural and production planning elements based on certain thresholds that are defined per producibility metric. Furthermore, the metrics relevant for eventually specified producibility scenarios are determined.

The concrete outputs of the execution phase are:

- List of critical architectural elements including the values of the producibility metrics that caused the classification as critical
- List of production planning elements including the values of the producibility metrics that caused the classification as critical
- Results regarding Producibility scenarios

The list of critical production planning elements is separated into a list of critical production work activities, production iterations, and resources.

The results of the execution phase are used by architects and production planners in the consolidation phase to consolidate the lists and derive recommendations.

In the following sub-section, the identification of critical elements based on the analysis algorithm is described.

5.3.1 Identification of Critical Elements

As mentioned above, the identification of potentially critical elements is based on the producibility views. Each producibility view refers to architectural elements and a certain type of production planning element, i.e., a production wok activity, a production iteration, or a resource. Consequently, a certain subset of the overall set of producibility metrics can be determined from each producibility view. Based on an evaluation of the values of the producibility metrics with respect to certain thresholds, architectural and production planning elements can be assigned to the lists of critical architectural and production planning elements while processing a certain producibility view.

The identification of potentially critical elements follows a specific analysis algorithm. One producibility view after the other is analyzed. The order of analyzing the producibility views in general is arbitrary. Typically, the production work activity view is available before the production iteration view, and the resource assignment view. Hence, we decided to process the production work activity view first, followed by the production iteration view, and the resource assignment view. In each analysis step, certain critical elements are identified and added to the respective lists of critical production work activities (PWA), critical production iterations (PI), critical resources (RES), and critical architectural elements (AE) Figure 55 provides an overview of the algorithm. The complete algorithm in pseudo-code notation can be found in Appendix B.

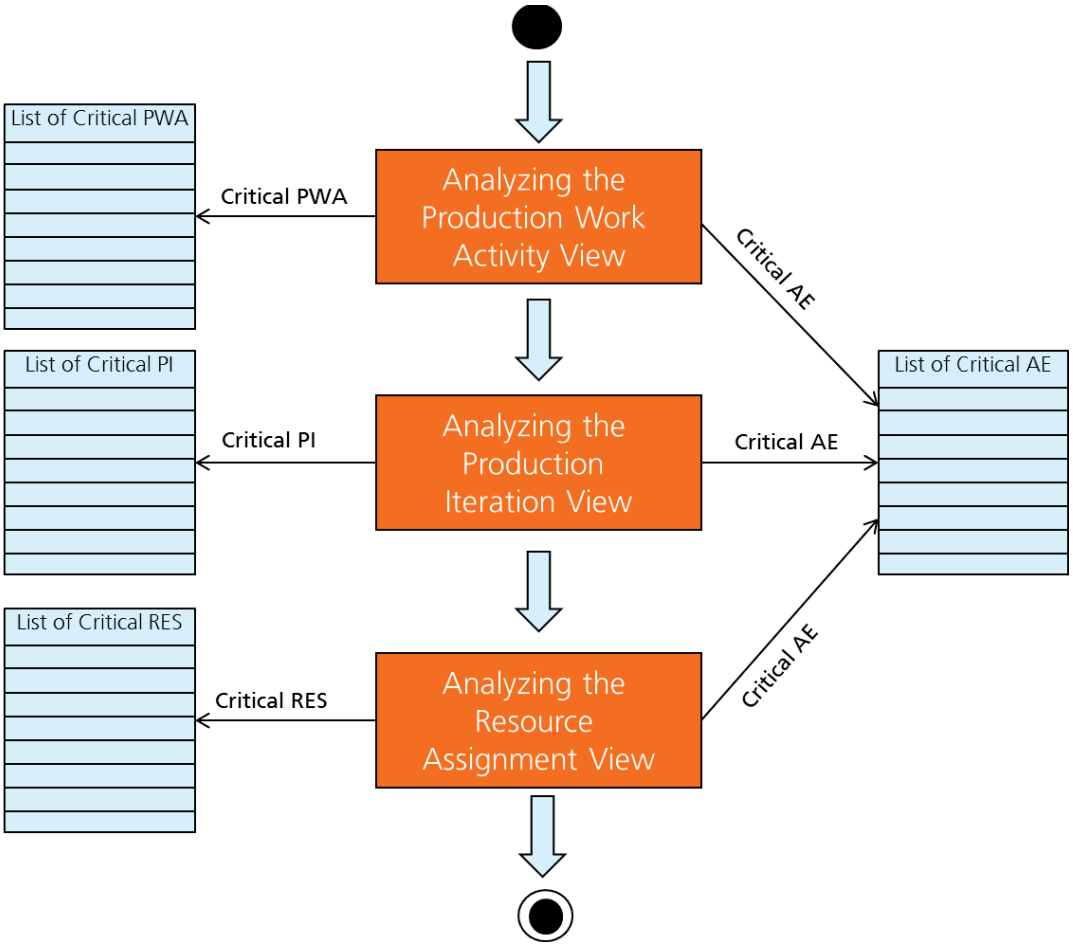


Figure 55: Overview Identification Algorithm

Table 4 shows the conditions/thresholds that are used by default to check, if an elements is supposed to be critical. If the condition in Table 4 is evaluated to true, the respective element is classified as critical. The definition of the conditions followed a rather pessimistic strategy. The thresholds when an element is classified as critical are rather low. If we take as an example #PWA_producing(AE) > 1, this condition is evaluated to true as soon as #PWA_producing(AE) does not have its optimal value. But there are cases where already a value of #PWA_producing(AE) = 2 is critical, for instance, if two different teams that are geographically distributed work on the AE. Hence, it is important to use the context factors and the checklists presented in Section 5.4.2 to consolidate if the element is really critical.

Producibility Metric	Condition checked
#PWA_producing(AE)	>1
#PWA_consuming(AE)	>0
#AE_produced_by(PWA)	>1
#AE_consumed_by(PWA)	>1
Coupling(PWA)	>0

#Shared_AE(PWA)	>0
#PI_producing(AE)	>1
ProductionDuration(AE)	>1
#AE_involved_in(PI)	> round(| AE_{all}| / |PI_{all}|)
Coupling (PI)	>0
#Shared_AE(PI)	>0
%Shared_AE (PI)	>0
%Completed_AE_after(PI)	< IterationNumber(PI) / |PI_{all}|
%Created_AE_after(PI)	< IterationNumber(PI) / |PI_{all}|
#Resources_working_on (AE)	>1
#AE_worked_on_by(Res)	> round(| AE_{all}| / |RES_{all}|)
Coupling(Res)	>0
#Shared_AE(Res)	>0

Table 4: Conditions for identifying critical Elements

The following functions are used in Table 4 and need further explanation:

- round() determines the next larger integer number based on a real number provided as input

- |{AE}| is the number of elements in the set {AE}

- IterationNumber(PI) is an integer number representing the production iteration PI. The production iterations are numbered sequentially, i.e. 1,2,3…

Table 4 can be found again in Appendix A.

Example

In the following, an example for performing the identification of critical elements based on a given set of producibility views is presented.

The system is supposed to be built in three production iterations with three teams involved overall. The modeling of producibility views has been performed based on inputs from architects and production planners. The resulting production iteration view is shown in Figure 56. The resource assignment views is shown in Figure 56. The inputs of architects and production planners the example is based on can be found in Appendix D.

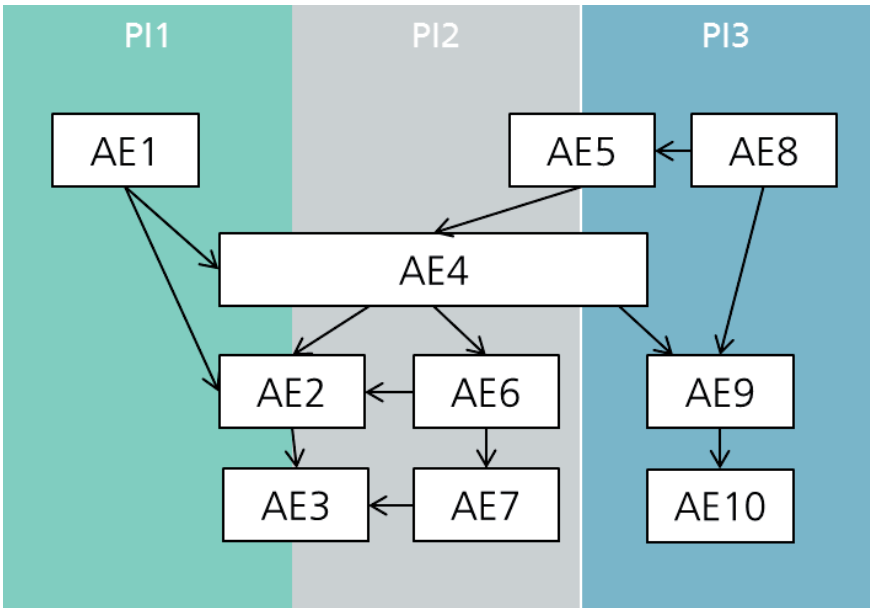


Figure 56: Example - Production Iteration View

If we apply the analysis algorithm to the producibility views modeled in Section 5.2.2, we get the lists of critical elements shown in Table 5, Table 6, Table 7, and Table 8 as a result.

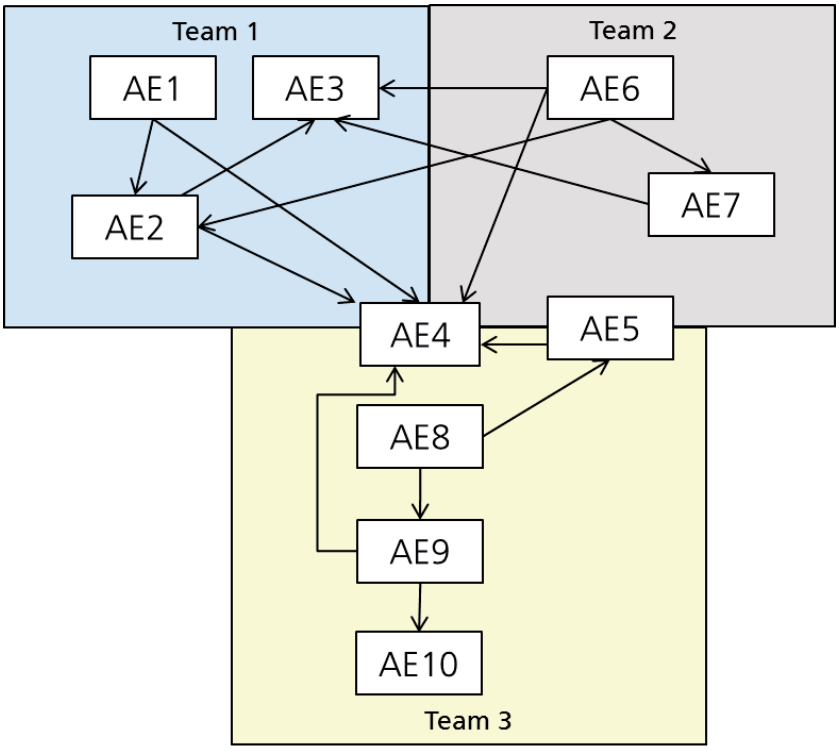


Figure 57: Example - Resource Assignment View

Critical AE	#PWA_producing	#PWA_consuming	#PI_producing	ProductionDuration	#Resources_working_on	CriticalDimensions
AE2	2	5	2	2	1	2
AE3	3	1	2	2	1	2
AE4	3	3	3	3	3	3
AE5	2	1	2	2	2	3

Table 5: Example - List of Critical Architectural Elements

Critical PWA	#AE_produced_by	#AE_consumed_by	Coupling	#Shared_AE	%Shared_AE
PWA1	1	2	4	0	0
PWA2	2	0	6	2	100
PWA3	3	2	8	3	100
PWA4	1	3	7	1	100
PWA5	1	1	4	1	100
PWA8	2	2	2	1	50
PWA9	2	1	5	2	100

Table 6: Example - List of Critical Production Work Activities

Critical PI	#AE_involved_in	Coupling	#Shared_AE	%Shared_AE	%Completed_AE_after	%Created_AE_after
PI1	4	0	3	75	10	40
PI2	6	0	4	2/3	50	70
PI3	5	0	2	40	100	100

Table 7: Example - List of Critical Production Iterations

Critical RES	#AE_worked_on_by	Coupling	#Shared_AE	%Shared_AE
Team1	4	2	1	25
Team2	4	2	2	50
Team3	5	2	2	40

Table 8: Example - List of Critical Resources

In each table, the cells representing a value that makes the related condition in Table 4 to be true is marked in grey. This is helpful for architects and production planners to start the interpretation of the results of the execution phase in the consolidation phase.

Besides information on single architectural and production planning elements, the execution phase comes up with results regarding the producibility scenarios eventually modeled in the preparation phase. The following sub-section describes the execution phase deals with the producibility scenarios.

5.3.2 Analysis of Producibility Scenarios

As mentioned in Section 5.2.1, producibility scenarios are used to specify requirements into production from different stakeholders. The producibility analysis evaluates, if the producibility scenarios are likely to be fulfilled. In the mapping step of producibility scenarios described in Section 5.2.3, relationships from a producibility scenario to producibility views are established. Hence, architectural elements and production planning elements involved in the respective producibility scenarios are identified.

The execution phase of the producibility analysis makes two contributions towards analyzing producibility scenarios.

The execution phase provides results regarding the producibility metrics related to producibility scenarios defined in Chapter 4. These producibility metrics are:

- #PWA_producing{AE}
- #PI_involving({AE})
- PI_finishing({AE})
- #Resources_working_on({AE})

The producibility metrics that are relevant for a specific producibility scenario depend on the producibility dimension that is addressed by the scenario. #PWA_producing({AE}) is relevant for the alignment of architecture and production work breakdown structure. #PI_involving({AE}) and PI_finishing({AE}) are relevant for the alignment of architecture and production schedule. #Resources_working_on({AE}) is relevant for the alignment of architecture and resource assignments.

Besides determining producibility metrics related to producibility scenarios, the execution phase makes a second contribution regarding producibility scenarios. Information from the identification of critical elements described in Section 5.3.1 that is relevant for a specific producibility scenario is attached to it. If, for instance, an architectural element that is involved in a producibility scenario is classified as critical, this information is attached to the producibility scenario.

Table 9 shows an example of an extended version of the template for documenting producibility scenarios. Additional rows have been added with producibility metrics relevant in the context of the producibility scenario. The information in the table is the starting point to analyze the producibility scenario in the consolidation phase.

Name	Name of the scenario
Stakeholder	The main stakeholder interested in this scenario.
Producibility Dimension	Dimension selected from the quality model of producibility (alignment of architecture and production WBS, alignment of architecture and production schedule, alignment of architecture and resource assignments)
Description	Description of the scenario
Involved AE	The instances of architectural elements referenced in the scenario
Involved PPE	The instances of production planning elements (e.g. production work activities, production iterations, resources) referenced in the scenario
#PWA_producing({AE})	The number of PWAs involved in producing the set of AEs involved in the producibility scenario
#PI_involving({Involved AE})	The number of PI involved in producing the set of AEs involved in the producibility scenario
PI_finishing({Involved AE})	The ID of the PI where all AEs involved in the producibility scenario are completed.
#Resources_working_on({Involved AE})	The number of resources involved in producing the AEs involved in the producibility scenario.

#Critical AE involved	The number of critical AE involved in the producibility scenario.
#Critical PWA involved	The number of critical PWA involved in the producibility scenario.
#Critical PI involved	The number of critical PI involved in the producibility scenario.
#Critical RES involved	The number of critical RES involved in the producibility scenario.

Table 9: Example for extended Producibility Scenario Template

5.4 Consolidation Phase

In the consolidation phase of the producibility analysis, the results of the execution phase are analyzed by architects and production planners, and recommendations on how to change the architecture or the production plan are derived to increase the producibility of the system. As mentioned above, the input to the consolidation phase for architects and production planners are the lists of critical elements derived in the execution phase.

The consolidation phase is conducted in three steps as shown in Figure 58.

In a first step, architects and production planners check the received lists for completeness and extend them if required. The reason for eventually

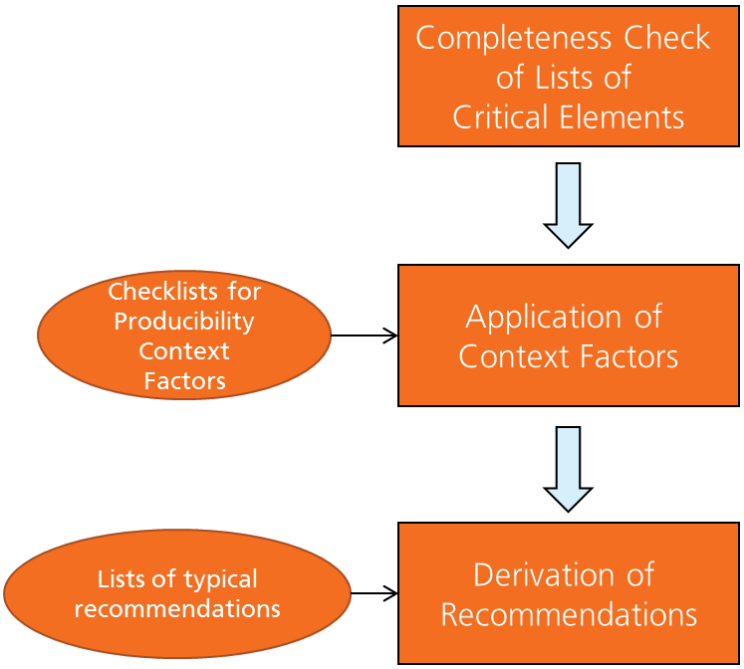


Figure 58: Steps of the Consolidation Phase

required extensions is, that the algorithm used in the execution phase detects critical architectural and production planning elements based on their relationships and not based on properties of single architectural or production planning elements. If an architectural elements is supposed to be quite complex because of the algorithms to be implemented inside, for instance, this might be a reason to classify the architectural element as critical. Architects and production planners can add elements based on their personal experience in this step.

In a second step, the context factors of the quality model of producibility (see Section 4.5) are applied to the lists of critical elements. Therefore, the context factors have been used to create checklists with questions regarding such context factors. All questions of the checklist can be answered with yes or no. Each question that is answered with yes reduces the risk that the related element causes production problems.

In a third step, recommendations are derived by architects and production planners that reduce the risk of production problems caused by the critical elements identified. This step is conducted jointly by architects and production planners. The reason for the get together is that changes made to the architecture or the production plan influence each other and need to be aligned to each other. Furthermore, this step again enforces the communication between architects and production planners. This final step of the consolidation phase is supported by a list of typical options for recommendations.

The concrete outputs of the consolidation phase are:

- Consolidated lists of critical elements
- List of recommendations

The following section discusses the steps of the consolidation phase in more detail.

5.4.1 Completeness Check of List of Critical Elements

Architects and production planners check the lists of critical elements for completeness in this step based on their experience. The reason for the check for completeness is that the algorithm used to identify critical elements can only consider the information modeled in the producibility views. Properties of, for instance, architectural elements like internal complexity or experiences made with similar architectural elements in the past are not included in the model but should lead to the decision to classify an architectural element as critical.

Table 10 shows examples of architectural elements that should be considered as critical and that cannot be detected by the algorithm used in the execution phase.

Potentially Critical Architectural Elements
AEs that are complex because of algorithms, data structures, etc.
AEs with high quality requirements regarding security, performance, availability, flexibility, etc.
AEs using legacy technology or any technology where little expertise is available in the organization
AEs involved in realizing complex connectors
AEs that are provided by external suppliers
…

Table 10: Potentially Critical Architectural Elements

Complex architectural elements bear the risk of causing delays, effort overhead, or low quality. Realization and testing of complex algorithms, for instance, might cause delays or effort overhead. The same holds true for architectural elements with high quality requirements. Technologies might cause production problems, especially if little experience on the technologies are available in the organization or if only few experts are available at all like in the case of legacy technologies. Connectors are often a source of production problems because of their inherent complexity. Architectural elements of external suppliers should be considered critical, for instance, if the supplier is not yet well known to an organization.

Potentially Critical Production Iterations
First PI in a project
Last PI in a project
PI involving complex AE
PI involving external resources for the first time
PI performed during special season
…

Table 11: Potentially Critical Production Iterations

Table 11 shows potentially critical production iterations. The first and the last production iteration are often critical. The first one because the teams need to get started and the last one because the project needs to get finished and all issues remaining from previous production iterations need to be addressed in addition. Production iterations involving complex architectural elements might be a problem, as well as production iterations involving external suppliers or teams for the first time. There can even be seasonal reasons for classifying a production iteration as critical, for instance, if it is supposed to be performed during the holiday season or during flu season.

Potentially Critical Resources
Resources located far abroad from core team
External suppliers
International teams with cultural differences
…

Table 12: Potentially Critical Resources

Table 12 shows potentially critical resources. Resources might be supposed critical if they are located far abroad the core team. External suppliers need to be handled with specific care especially if they are not yet well known. International teams involving different cultures can be a source of issues as well.

After architects and production planners have considered additional critical elements, the lists are consolidated with respect to the context factors defined in the quality model of producibility.

5.4.2 Application of Context Factors

In this step, context factors are applied to the lists of critical elements. As mentioned above, checklists containing questions derived from the context factors are used in this case. All questions can be answered with yes or no. The questions are formulated in a way that each question that is answered with yes reduced the risk of production problems caused by the respective element. After the context factors have been applied to the respective critical elements, architects and production planners have to decide if they still consider an element as critical or if the context factors mitigate the production risks from their point of view. This decision is not taken automatically but requires human judgment.

Table 13 shows as an example the checklist for critical architectural elements. The checklist consists of a section with general questions that should be asked in the case of each critical architectural element. In addition, it contains questions that should be asked in case the respective producibility metric exceeded the defined threshold.

General questions for each AE:

- Is the quality of the architecture documentation with respect to the AE high?
- Is a production work activity type, i.e. a guideline describing how to produce the AE available?
- Are certain development activity types supporting the production of the AE, for instance, continuous integration, regression testing, generation of parts of the AE, etc.?
- Are tools providing specific support for the production of the AE?
- Can the AE be built based on reuse and is a process how to reuse attached to the reusable artifacts?
- Are the resources providing and potentially adapting the reusable artifacts available when the AE is supposed to be produced?

- Is the available team experienced with the technologies used to realize the AE?
- Is the AE produced by internal resources and are they co-located?
- If the AE is produced by external resources or internal resources that are not co-located, is an appropriate communication infrastructure in place, and an infrastructure that facilitates the exchange of artifacts?
- Are contact persons for the AE under analysis and for all related AE in place that can help in solving issues?

#PWA_producing(AE)

- Is a certain order defined for performing the PWAs?
- Are the PWAs performed by the same team (or by co-located teams)?
- Is the internal design of the AE prepared for parallel work and/or incremental extension?
- Are integration and test processes defined for the AE?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?
- Does the tool infrastructure support parallel production well?

#PWA_consuming(AE)

- Are the PWAs consuming the AE produced later on?
- Are the consuming PWAs performed by co-located resources?
- If the resources producing consuming PWAs are not co-located, are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Do the involved resources know each other personally?
- Is the quality of the architecture documentation high, especially the documentation of the interfaces of the AE?

#PI_producing(AE)

- Are the same resources producing the AE throughout all iterations?
- Is the internal design of the AE prepared for incremental extension?
- Are integration and test processes defined for the AE?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?

#Resources_working_on(AE)

- Are the resources producing the AE co-located?
- Are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Do the involved resources know each other personally?
- Does each involved team have one single point of contact, for instance, a chief programmer?
- Do the resources work on parts of the AE separated in the design of the AE?
- Is one resource responsible for integration, final test, and deployment of the overall AE?

Table 13: Checklist for Architectural Elements

Similar checklists for production work activities, production iterations, and resource assignments exist and that can be used by production planners can be found in Appendix C.

Context Factors and Producibility Scenarios

The application of context factors to the lists of critical elements also affects the producibility scenarios. If certain elements are removed from the lists of critical elements, this affects the producibility scenario as potentially less critical elements are involved. The documentation of the producibility scenario has to be adapted in this case.

The application of the checklists are a good preparation for deriving recommendations in the next step. If certain answers related to critical elements are answered with no, for instance, if no test and deployment processes are defined for an architectural elements that is often modified, architects could recommend to invest in the definition of test and deployment processes. In the following section, it is described how recommendations can be derived to mitigate the risk of production problems caused by the critical elements.

5.4.3 Derivation of Recommendations

The final step of the consolidation phase is the derivation of recommendations. After the lists of critical elements that came out of the execution phase has been checked for completeness and the producibility context factors have been applied, architects and production planners have to decide which measures to take to prevent potential production problems.

Architects and production planners have certain options for recommendations they can realize in their respective area of responsibility. Architects can change, for instance, the overall functional decomposition of the system or perform local changes by, for instance, splitting single architectural elements. Production planners can change the production work breakdown structure, the assignment of production work activities to production iterations, the assignments of resources, etc. Furthermore, both, architects and production planners can try to influence the context factors of the quality model of producibility to prevent production problems caused by critical elements. From an overall project perspective, architects and production planners jointly should identify the measures that lead to an improvement of producibility which often are a combination of actions to be performed on the architecture and actions to be performed on the production plan. Hence, architects and production planners conduct this last step of the consolidation phase together.

Nevertheless, it is beneficial to provide a general overview of potential recommendations from an architect's and a production planner's point of view to show the available options to them as a starting point for their joint discussion.

The following tables provide an overview of possible recommendations. They are not complete in a sense that no other options can be imagined.

But they provide a good starting point for the derivation of recommendations or measures to increase the producibility in a concrete case.

Table 14 shows potential recommendations for critical architectural elements.

Recommendations regarding Architectural Elements
Split AE
Improve internal design of the AE to support parallel work and/or incremental extension
Try to produce AE based on reuse
Ask production planner to reduce number of involved resources producing the AE by changing resource assignments
Ask production planner to reduce number of PWAs producing the AE
Ask production planner to reduce number of production iterations producing the AE
Improve tool support to produce the AE
Define production work activity type to support production of the AE
Define development activities specifically supporting the production of the AE
Improve the documentation of the AE
…

Table 14: Recommendations regarding Architectural Elements

Table 15 shows potential recommendations regarding production iterations.

Recommendations regarding Production Iterations
Move AEs or PWAs to other PIs
Extend duration of PI, plan time buffer at the end of the iteration
Involve different resources in the PI
Reduce number of involved resources to save communication effort
Improve support for producing involved AE, e.g. define production work activity types, improve tool support, etc.
Ask architect for splitting shared AE if possible
Ask architect for reducing coupling between AEs
…

Table 15: Recommendations regarding Production Iterations

Table 16 shows potential recommendations regarding resources.

Recommendations regarding Resources
Reduce AEs produced by resource
Clearly assign AEs to resources, prevent sharing of AEs
Improve communication between resources

Change team composition
Ask architect for splitting shared AE if possible
Ask architect for reducing coupling between AEs
Teach resources in used technologies
…

Table 16: Recommendations regarding Resources

Table 17 shows potential recommendations regarding production work activities.

Recommendations regarding Production Work Activities
Change work breakdown structure
Reduce number of involved resources to save communication effort
Improve support for producing involved AE, e.g. define production work activity types, improve tool support, etc.
Ask architect for splitting shared AE if possible
Ask architect for reducing coupling between AEs
Ask architect to change functional decomposition with respect to the PWAs
…

Table 17: Recommendations regarding Production Work Activities

Example

In the example used in Section 5.3.1, the following recommendations could help to prevent potential production problems. The solutions chosen here directly influence the respective producibility metrics:

- Split AE4: The architectural element AE4 is split into three parts. One of the three parts is assigned to each production iteration.
- Move AE2: AE2 is moved completely to PI1, i.e. the functionality supposed to be added in PI2 in the original plan is already completely realized in PI1.
- Move AE3: AE3 is moved completely to PI1 with the same argumentation than in the case of AE2.
- Move AE5 and change resource assignment: AE5 is completely moved to PI2 and assigned to Team 2.

The recommendations have several positive effects. The overlapping between the production iterations is eliminated (see Figure 59), i.e. no architectural elements are shared anymore between production iterations. Furthermore, no architectural elements are shared between teams.

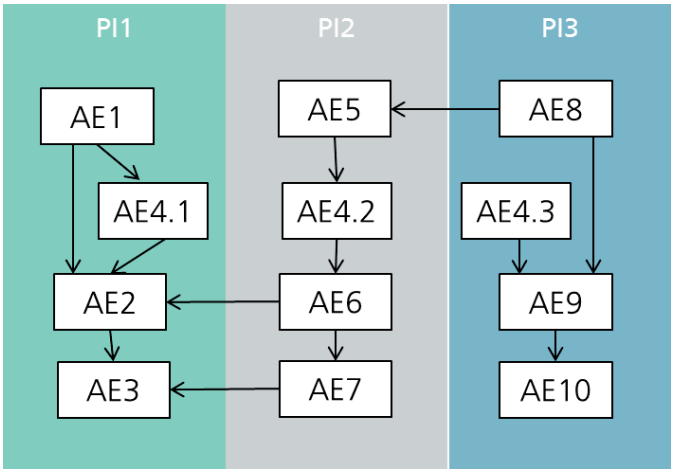


Figure 59: Improved Production Iteration View

The architectural element AE4 is no longer modified in each production iteration and by different teams. The splitting might lead to a certain amount of redundancy in the resulting three architectural elements AE4.1, AE4.2 and AE4.3. As an alternative to splitting, it eventually would have been possible to try to improve the internal structure of AE4 in a way that the modification performed over time mainly affect separate parts. This measure could have been supported in addition by defining regression tests in each production iteration to make sure functionality is not broken later on, and to take specific care of documenting the architectural element. The architectural elements AE2, AE3, and AE5 have been moved in addition and AE5 has been completely assigned to

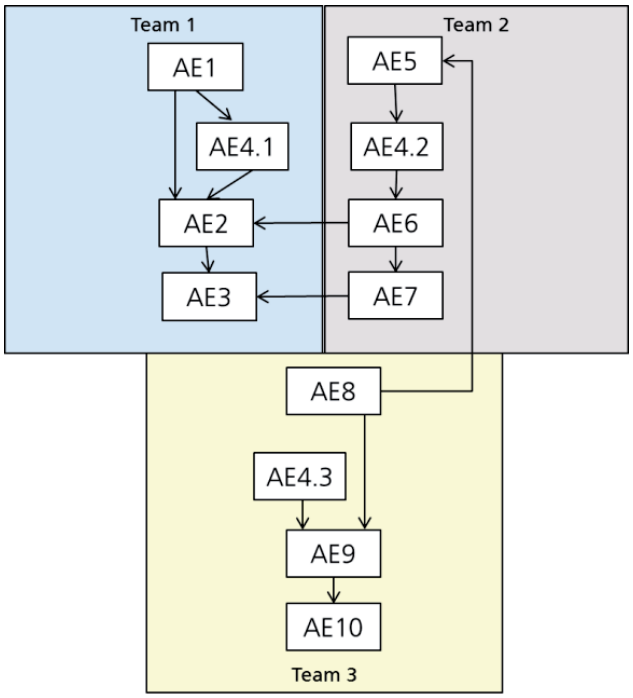


Figure 60: Improved Resource Assignment View

Team 2. Figure 60 shows the improved resource assignment view.

In the end, each production iteration produces four architectural elements completely, which leads to a sustainable growth of the system over time. The risk of breaking functionality as architectural elements are modified several times has been reduced. Especially, production iteration PI2 benefits from the recommendations. Less architectural elements are involved and architectural elements are no longer shared with other production iterations. The risk of not finishing PI2 in time or the risk of effort overhead in PI2 has been reduced.

Recommendations regarding Producibility Scenarios

Architects and production planners finally have to make conclusions regarding the producibility scenarios. The recommendations to address the criticality of architectural and production planning elements influence the producibility scenarios as the number of critical elements involved in a producibility scenario might be reduced. Nevertheless, it must be decided to which degree a producibility scenario is fulfilled and if additional recommendations need to be formulated.

Similar to architecture evaluations, the decision if and to which degree a scenario is fulfilled needs to be taken by an expert and can hardly be automated. The documentation of a producibility scenario as shown in Table 9 provides facts that can be used by architects and production planners to take a decision. If architects and production planners decide that a producibility scenario is not fulfilled, they have again the options for recommendations introduced in Table 14, Table 15, Table 16, and Table 17 as a guideline.

Limited Support for Derivation of Recommendations

The current support for deriving recommendations is limited. The focus of this work is to support the analysis of producibility, i.e., the identification of critical architectural and production planning elements. The lists of recommendations are not meant to be complete and they are not yet validated. A more comprehensive support for deriving recommendations requires a better understanding of the process of design for producibility, i.e., how to guarantee producibility by construction. This process is not considered in detail in this dissertation but left open for future work.

In the following section, the available tool support is presented.

5.5 Tool Support

The producibility analysis is supported by a tool prototype. The tool prototype is realized as an extension of Enterprise Architect [EA11]. Enterprise Architect (EA) is a professional modeling tool broadly used in industrial practice. The two main features of the tool prototype supporting the producibility analysis are:

Modeling Producibility Views

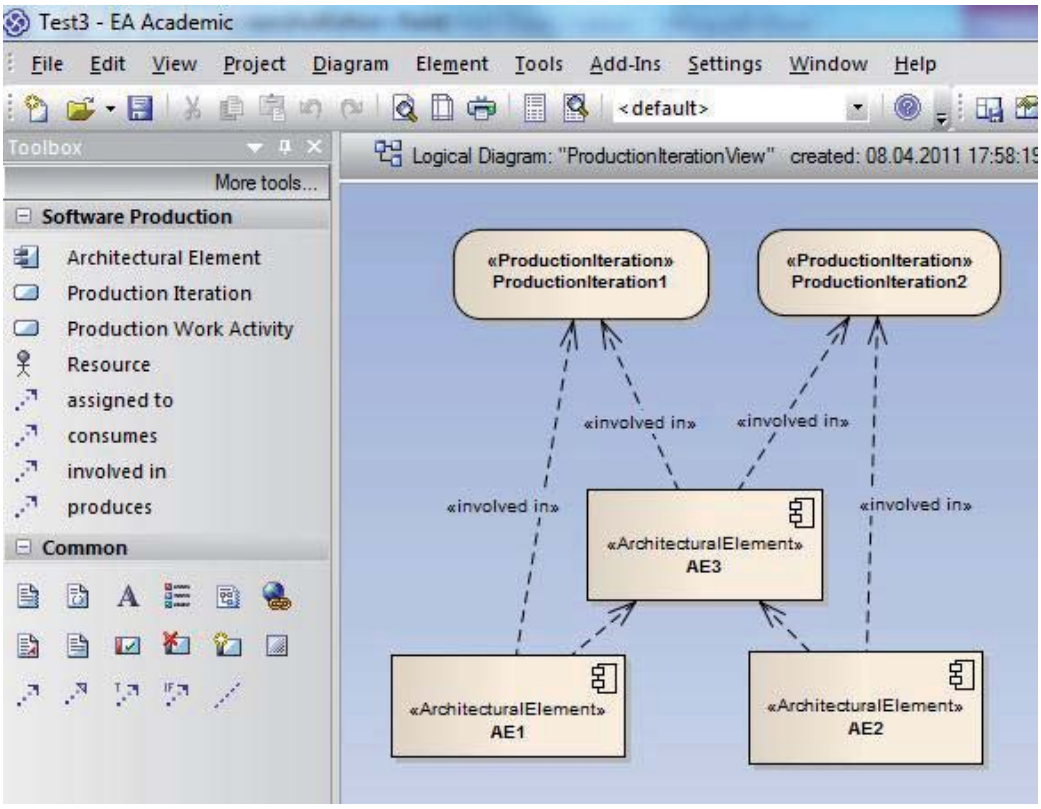


Figure 61: Modeling Producibility Views in EA

– Support for modeling producibility views: All elements required to model producibility views have been integrated into Enterprise Architect. Hence, models of an architecture can now be complemented by production work activity views, production iteration views, and resource assignment views.
– Determination of producibility metrics: The producibility metrics that have been defined in the quality model of producibility can be determined based on the modeled producibility views.

In the following, the two main features that have been realized are described in more detail.

Figure 61 shows how the elements required to model producibility views have been integrated into Enterprise Architect. The so-called toolbox of the Enterprise Architect has been extended by a software production tool. The software production tool contains the elements used in software production like architectural elements, production iteration, etc. The can be added via drag & drop to the main working area in the middle of the screen to model the respective producibility views. Thereby, producibility views can be added to the overall model via the project browser shown on the right.

Technically, the software production tool to model producibility views is realized by means of a UML profile. The meta model of software production that contains all required modeling elements has been partially

Determination of Producibility Metrics

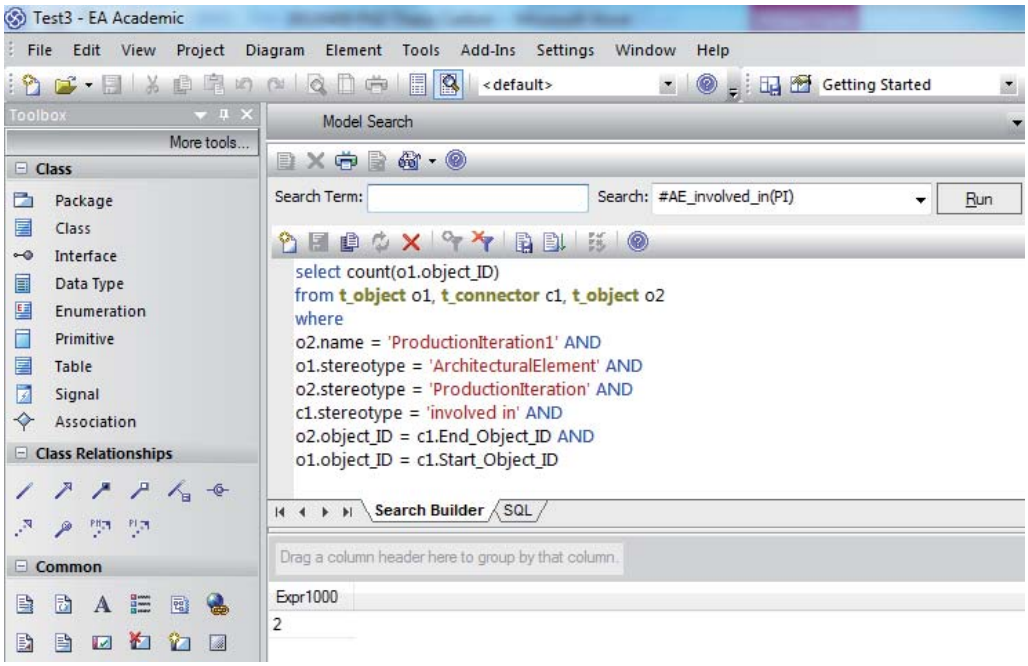


Figure 62: SQL Model Query in EA

modeled in Enterprise Architect as a UML profile that now can be imported to each project that wants to model producibility views.

The producibility metrics of the quality model of producibility can be automatically determined based on the producibility views modeled in EA. Technically, the metric determination uses the feature of EA to formulate SQL queries [SQL] on the underlying model. EA stores the models in an underlying SQL database. By means of an EA internal SQL model editor, the SQL queries are formulated. Queries can be stored, which enables us to formulate queries for each producibility metric, store them, and reuse them later on. Figure 62 shows, how metrics can be formulated in the SQL editor of EA.

The tool prototype shows the technical feasibility of modeling producibility views and determining producibility metrics based on such models. However, the tool support for the producibility analysis method should be extended in the future, for instance, to enable the modeling of context factors as properties of the modeling elements in the software production tool in EA.

6 Validation

In this chapter, we present the existing validation results of this thesis and give an outlook on future validation activities. The documentation of the validation results is twofold. First, we present the results of a series of industry and applied research projects that have been conducted throughout the course of this thesis. These projects supported in elaborating the contributions of this thesis, i.e. mainly the meta-model of software production, the quality model of producibility, and the producibility analysis method, and in many cases contributed to their initial validation. Second, we present a case study that has been conducted on the final version of the producibility analysis method.

6.1 Projects Accompanying this Thesis

In this section, we describe the results of a series of industry and applied research projects that accompanied this thesis, contributed to elaborate the results and served as initial validation.

6.1.1 Project "Virtual Office of the Future"

The probably most influencing project for this thesis is called "Virtual Office of the Future" (VOF). It has been conducted from 2003-2008 together with RICOH Co. Ltd., a Japanese manufacturer of office devices like so-called Multi-Functional Office Peripherals (MFP). The general project setting was that Fraunhofer IESE conducted research on Software Engineering for so-called virtual office environments inspired by the practical problems of Ricoh, and transfers the research results to Ricoh's MFP business unit.

RICOH's goal for the near future is to sell integrated office environments, i.e. not only office devices, but overall office infrastructures supporting the office workflows of their customers with Ricoh devices well integrated. Hence, they required a Software Engineering methodology that enables them to develop such systems. The key solution idea in the beginning of the project was to adopt product line technology to build office environments based on reuse. Hence, we analyzed typical workflows of RICOH customers and created a reference architecture for virtual office environments [CJK+08]. The major issue has been the processes that enable RICOH to build office environments based on the reference architecture. They wanted to get concrete guidance on how to build office environments. Hence, we developed the idea to provide them with pro-

cesses similar to the ones they know from producing their MFPs, which was the initial idea for software production processes. We defined the general strategy to set-up office environments with a customer, which is to adopt and iterative approach and realize a set of workflow or workflow variants in each iteration [CJM+08]. We provided guidance on how to produce workflows and related architectural elements like services, backend adapters, etc. In other words, we defined production work activities and high level production plans guiding engineers in building workflows and related architectural elements. Hence, the VOF project provided us with initial experience on software production as we defined an architecture and production plans and processes tailored to it.

The VOF project also provided input to the definition of the quality model of producibility. We learned about the importance to set-up the architecture according to the iterations defined in the production plan. The idea of software production will not work well if each iteration again modifies parts of the system that have already been created and delivered. As the production of office environments was planned to be conducted in a distributed setting by Ricoh, it was important to assign the available resources to different parts of the architecture and define the architecture in a way to enable this.

Overall, the feedback by Ricoh was very positive. Ricoh expects to be able to deliver integrated office environments with short time to market which is enabled by the concepts of software production [CAU+09]. They planned to build a production environment with tools supporting the defined production processes and development frameworks that enable them to easily create the architectural elements being part of the reference architecture.

6.1.2 Projects in the Airline Management Domain

In 2008 and 2009, we were involved in a series of projects with a customer in the airline management domain. Our tasks were to assess architectures of large workflow-based information systems supporting, for instance, booking or check-in. Based on our assessments, we also were involved in designing new architectures for the next generation of such systems and supported in developing strategies for migration.

Especially, when we were involved in designing future architectures and planning their realization respectively migration, the idea of software production again came into play. Similar to the office environments in the VOF project, the production or migration of workflows has been a recurring pattern and it seemed promising to set-up production processes for workflows and related architectural elements and design the architecture in a way to support the iterative production and deployment of workflows. Based on our argumentation on the relationship of architecture, production plan, and production processes we got involved not on-

ly into architectural design, but also into production planning and production process design. Unfortunately, the overall project on the customer side was cancelled and consequently our involvement, too.

Nevertheless, we were able to collect several important lessons learned. The idea of software production is applicable in the context of single but large workflow-based information systems. We were involved in planning production and defining production processes. Unfortunately, the results have not been used in this case.

The projects again provided input for the quality model of producibility. We learned about the importance of considering the technology mix used in a project, especially if legacy systems are involved. Several external providers have been involved in the project, which helped us to understand the importance of understanding their capabilities in the context of software production.

6.1.3 Project "ProKMU"

In the applied research project ProKMU ("Produktlebenszyklusmanagement in KMU (ProKMU): Methodische Unterstützung für die kostengünstige Implementierung und Anpassung interdisziplinärer, kooperativer, flexibler, PLM-Lösungen"), funded by „Bundesministeriums für Wirtschaft und Technologie (BMWi) - Zentrales Innovationsprogramm Mittelstand" (ZIM), we developed a method to customize product data management systems. Product data management systems are used by manufacturers, for instance, in the automotive industry, to manage bills of materials for their products, tracking exactly which parts have been used in which product, etc. The systems are characterized by complex data structures. Workflows play a minor role. Such systems typically need to be customized to the data formats used in organizations, they need to be integrated with other systems supporting the manufacturing process, etc.

The project gave us the chance to apply the idea of software production to information systems that are less workflow-oriented than the ones described in Section 6.1.1 and Section 6.1.2. The major architectural elements that need to be produced in this case are data structures, services accessing such data structures, and editors that enable the users of the system to view and manipulate data. We defined a customization process and reference plans for certain types of customizations together with an industrial partner in the project that are inspired by the idea of software production. Typical steps of the process are the creation and customization of data structures, services, and editors. The industrial partner even has built tools to support the production of data structures, services, and editors by means of generative approaches.

The project showed that the idea of software production is also applicable in the context of data-centric information systems. We received further input for the quality model of producibility. Compared to the projects introduced in the previous sections, we specifically learned about the potential of tools to support production.

6.2 Case Study: Mobile Configuration Assistant

The producibility analysis method has been evaluated in an industrial case study. This section describes the goals of the case study, the context, the approach that has been chosen to conduct the case study, the results, and the threats to validity in detail.

6.2.1 Goals

The goals of the evaluation conducted in the context of the case study are to validate the hypothesis stated in Section 1.5:

H1 – Effectiveness of the Producibility Analysis Method with respect to Time and Effort: *The producibility analysis method reduces time and effort spent on production (i.e., in this case the set of all activities conducted after architectural design and project or production planning) by at least 25%.*

H2 – Completeness of the Identification of Critical Elements: *The producibility analysis method detects at least 75% of critical elements (including architectural and project or production planning elements).*

H3 – Correctness of the Identification of Critical Elements: *At least 90% of the elements identified by the producibility analysis method as critical are really critical in the end, i.e. less than 10% of the identified elements are false positives and not causing any production problems.*

The case study was supposed to deliver initial empirical evidence to accept such hypotheses.

6.2.2 Context

Industrial Project

The case study has been performed in the context of an industrial project conducted by Fraunhofer IESE, the University of Kaiserslautern, and John Deere. John Deere is a manufacturer of systems for the agricultural domain consisting of machines like tractors or combines but also software and services sold with such machines.

John Deere Mobile Configuration Assistant

The project were the case study was conducted is called "Mobile Configuration Assistant" (MCA). The MCA is a mobile application running on a tablet device that is used by drivers of machines to configure their equipment. Machines like tractors and attached implements need to be configured for a specific task to be performed on the field like planting, spraying, or harvesting to deliver optimal performance. Today, the existing configuration solutions running on a display on the machine are not as usable as expected by the drivers, they are hard wired with the machine and cannot be used with other machines, and the software on the display is not easily extendible, for instance, if new configuration options are available. The goal of the project therefore was to develop a new configuration software on a mobile device that is highly usable also for novice users and easily maintainable by John Deere engineers. John Deere selected the iPad as the platform to realize the MCA. The goal was to develop a native iOS application. Other technologies required to realize the MCA were not prescribed by John Deere.

Project Team

The project team consisted of a team of twelve master students of the University of Kaiserslautern. For them, the project was a so-called Master Project in Software Engineering. They have to conduct one of such master projects if they focus on Software Engineering during their Master of Computer Science. The master project is a special one in the sense that a real customer provides the requirements for the system to be developed and is involved throughout the project until the results are delivered. The student team was an international team. Six of the twelve students are participating in the ERASMUS Mundus Program and strive for a European Master in Software Engineering. They spend the second year of their master program in Kaiserslautern. A team of University and Fraunhofer IESE employees that manage the overall project and provide support in all Software Engineering topics supervises the student team. Supervisors for requirements engineering, architecture and design, implementation, and quality assurance were available to them.

Project Approach

The project has been conducted in a period of three month from October 4 – December 20, 2010. The students have acquired the required skills to perform the projects in Software Engineering lectures before the project. Nevertheless, as a kind of refresher they were taught the requirements, the UI and interaction design, and the architectural design approach to be used in the project in the beginning of the master project in half day short tutorials. Furthermore, they got a tutorial in iOS development as none of the students had previous experience in iOS development. As Fraunhofer IESE conducted iOS development project with students before, a tailored tutorial enabling newbies with programming skills in other programming languages to develop iOS applications.

The students started with eliciting and analyzing John Deere's requirements based on an initial problem statement and video-conferences with US employees of John Deere and employees of the John Deere site in Kaiserslautern. After that they designed a user interface and interaction

concept and an architecture. Requirements engineering, UI and interaction design, as well as architectural design were conducted by means of Fraunhofer IESE's state of the art methods Task-oriented Requirements Engineering (TORE) and Fraunhofer Domain-Specific Architecture (DSSA). Project planning was conducted by the supervisors based on state of the art project planning approaches and their experience in various industry projects. The supervisors provided a work breakdown structure and a production schedule and assigned the students in a joint meeting to the production work activities. After architectural design has been conducted, production has been performed according to the architecture and the production plan.

6.2.3 Approach

Idea

The general idea of the case study approach is to perform a producibility analysis after architecture and production plan of the project are available. The producibility analysis is performed by a researcher that is not a member of the project team. The results of the producibility analysis are not fed back to the project before it has ended. After the end of the project, critical architectural and production planning elements and related production problems are identified in a retrospective with the project team. The results of the retrospective are compared with the results of the producibility analysis and the effect that the producibility analysis would have had on the project is estimated and discussed with the project team. Hence, the project without producibility analysis is able to serve as a baseline that can be compared against a fictitious project with producibility analysis.

Procedure

In more detail, the case study followed the following procedure. A researcher, in this case the author of this thesis, received the materials required to conduct the producibility analysis after architectural design and production planning had been finished, i.e. mid of November. The researcher was involved in acquiring the project and knew the general problem to be solved, but was not involved in any more activities, especially not after the project had officially started on October 4, 2010.

The producibility analysis has been conducted, but as mentioned above, no results were fed back to the project team. No communication with the project team occurred before the end of the project. After the end of the project, each member of the student team as well as the supervisors for project planning and architecture have been interviewed individually for approximately 30 minutes by the researcher. Thereby, the researcher asked the student interviewee about his or her role in the project, especially during production. All interviewees were asked to report on critical architectural and production planning elements and related problems that occurred during the project. The identification of critical elements has been supported by a walkthrough of the architecture documentation and the production plan to increase the completeness of the results. Es-

pecially the critical elements the interviewee was in touch with were discussed in detail.

After the interviewees had provided their input on critical elements and production problems, the results of the producibility analysis were presented to them including recommendations on how to solve the predicted problems. The interviewees were asked to provide their opinions regarding the proposed recommendations. In addition, the interviewees were provided with information on the producibility views and were asked for feedback regarding the potential usefulness of such views during the project.

Based on the results of the producibility analysis and the project retrospective conducted via interviews the case study results presented in 6.2.4 have been prepared. In addition, the researcher was provided with the effort data that have been collected during the project and the real timeline of the project, i.e., for instance, when the planned production iterations really ended.

6.2.4 Results

6.2.4.1 Overview of Results

Feasibility of the Method

The adoption of the producibility analysis method in the case study was feasible. It was possible to apply the method to the artifacts provided as input, i.e. the architecture documentation of the system and the production plan. The producibility views required to conduct the producibility analysis have been derived successfully in the preparation phase. The producibility metrics have been determined and critical architectural and production planning elements have been derived, as well as potential production problems and recommendations. Overall, the application of the producibility analysis method took ~8 hours including reading the input documents, deriving the producibility views, determining the critical elements, and deriving potential problems and recommendations.

H2 - Completeness

The case study shows that a high completeness of identified critical elements can be achieved. Table 18 provides an overview per element type. All architectural elements that turned out to be critical during the project have been identified by the producibility analysis. Also all critical production work activities and production iterations have been predicted up-front. In the case of resources, 2/3 of the resources that turned out to be critical have been identified up-front. Hence, overall 91,67% of critical elements have been identified, which is beyond the value of 75% that has been mentioned in H2. This means, that the case study provides some evidence that a completeness of more than 75% can be achieved, and the hypothesis H2 might be accepted.

H3 - Correctness

The results regarding the correctness of the identification of critical elements are also very promising. As shown in Table 18, overall a correctness of 95% has been achieved. Only in the case of architectural elements, a false positive occurred. In all other cases, 100% of results were correct. Hence, the goal of achieving at least 90% correctness as stated in H3 has been achieved in the case study, which provides some evidence that H3 can be accepted.

	Completeness (H2)	**Correctness (H3)**
Architectural Elements	100%	80%
Production Work Activities	100%	100%
Production Iterations	100%	100%
Resources	66,67%	100%
Overall	91,67%	95%

Table 18: Overview Case Study Results

H1 - Effectiveness

The critical elements that have been identified by the producibility analysis method caused several production problems in the project. This has been detected based on the interviews performed with the project team. Production iteration 1, for instance, has been finished with a delay of 4 days. The students stated that without having the production problems they would have been able to finish production iteration 1 in time. Hence, if the predicted critical elements and the related production problems would have been addressed successfully up-front, there would have been the potential to save up to 4 days in production iteration 1. Thus, the overall duration of production iteration 1 would have been reduced from 14 to almost 10 days, which is a reduction by ~29%. The students reported in the interviews, that there would have been even more delays if experienced supervisors from Fraunhofer IESE would not have actively participated in the implementation activities during the last days of production iteration 1.

Unfortunately, the work on production iteration 2 could not start before production iteration 1 has been finished because all resources still have been involved in production iteration 1. Hence, an overall delay of the project of 4 days occurred which lead to the fact that the planned scope of the project could not be realized in the end as there was a hard deadline. Thus, we can conclude that there is some evidence that the producibility analysis method bears the potential to save more than 25% of time as it has been stated in H1.

H1 also refers to effort. The effort spent by the resources has been tracked throughout the project. Overall, 944 hours have been spent on production in the case study. 743 hours have been spent on production iteration 1, 201 hours on production iteration 2. The reason that only 201 hours have been spent on production iteration 2 is on the one hand the delay in production iteration 1, and on the other hand the overall

project deadline. By means of the producibility analysis results, it would have been possible to reduce the effort overhead caused by the delay in production iteration 1 and shift the saved effort spent to production iteration 2. We can assume based on the collected effort data, that the 4 additional days spent on production iteration 1 caused 241 hours of effort. If we save this effort in production iteration 1, which means a saving of ~32%, we can shift 201 hours into production iteration 2 (we do not shift the effort of the supervisors into production iteration 2).

Although we do not save effort overall as described above, we free up 32% of effort in one production iteration to spent them in another one. This gives us some evidence, that there is potential to save overall project effort by means of the producibility analysis method and that H1 eventually can be accepted from an effort point of view.

6.2.4.2 Identified Critical Architectural Elements

The structural view of the architecture of the system contains 10 architectural elements, in this case these are 10 components existing at runtime. All 10 architectural elements have been included in the producibility analysis. Overall, 5 out of 10 architectural elements have been identified as critical.

2 out of 10 architectural elements have been classified as critical as a result of adopting the algorithm for the identification of critical elements. Both architectural elements continuously undergo changes during the project, both are produced by two production work activities. One of them has an increased coupling in addition.

3 out of 10 architectural elements have been identified as critical based on the guidelines provided to identify critical architectural elements. One architectural element because it is the foundation to realize one of the highest prioritized features requested by the customer. Two other ones because they realize the communication between client and server and it was known up-front that different technologies have to be integrated that furthermore are not well-known to the team.

4 of the 5 architectural elements that have been identified as critical turned out to be critical during the project. They caused effort overhead and delays. No additional architectural elements caused problems in the project. Hence, in this case 100% of critical architectural elements have been identified up-front by the producibility analysis method, whereas one architectural element turned out to be not critical, although it has been classified as critical. Hence, one false positive occurred, which means that 80% of the results have been correct in this case.

6.2.4.3 Identified Critical Production Planning Elements

The production work breakdown structure in the case study contains 12 production work activities. The production work activities oriented at the architecture (not at features to be realized) and always refer to exactly one architectural element. 4 of the 12 production work activities have been identified as critical. The reasons are sharing architectural elements and high coupling. Context factors like the internal design of the shared architectural elements do not help to compensate the problem in this case.

All the identified production work activities turned out be critical in the project. They refer to architectural elements that have already been identified as critical before. Hence, 100% of the production work activities identified as critical turned out to be critical, which means that no false positives occurred in this case. Unfortunately, four production work activities that have not been identified by the producibility analysis method as critical turned out to be critical and were not finished during the project as a consequence of the delays caused by the other production work activities. The students reported that they would have been able to finish the production work activities if they would have had more time. Hence, we can somehow assume that they would not have been critical, if the problems with the previous production work activities would have been solved. Thus, we somehow can argue that the producibility analysis identified all critical production work activities.

The production phase of the project was planned for four weeks. The customer required an intermediate presentation of project results showing that certain key features are already realized for the third week. Hence, two production iterations have been planned, the first one ending after two weeks delivering the system to be presented at the intermediate presentation. The second one ending after four weeks delivering the final version of the system. The first of the planned production iterations has been identified as critical. It creates more than 50% of the overall number of architectural elements. 5 of the 6 architectural elements produced in the iteration have been classified as critical before.

The students have been organized in three teams involved throughout the two production iterations. Each team takes care of one layer of the system, i.e. one team realizes the graphical user interface (GUI), one the business logic on the client side, and one the server side. Two of the teams have been classified as critical by the producibility analysis method, the team responsible for the business logic on the client and the server team. The business logic team has an increased communication effort with the other teams because of the coupling between the architectural elements assigned to them. The server team has to cope with 6 out of 10 architectural elements of the system but is not larger than the other teams. In the project, all three teams contributed to the production problems that occurred. All teams delivered their results too late in the

first production iteration. This means the producibility analysis method determined 2/3 of the overall number of critical resources and there were no false positives in this case.

Appendix E contains the values of the producibility metrics for all architectural elements, production work activities, production iterations, and resources considered in the case study.

In the following section, threats to validity that need to be considered while interpreting the results are discussed.

6.2.5 Threats to Validity

In this section, we discuss threats to validity according to the classification referred to in [WRH+00].

Conclusion Validity

The MCA case study provides only one data point as a basis to draw conclusions on accepting the hypotheses H1, H2, and H3, which is a serious threat to conclusion validity. Therefore, we tried to make sure that the collected effort data are highly reliable and that the interviews are conducted in a structured way to guarantee high quality of the interview results. The quality of the effort data was addressed by providing a standardized effort collection sheet to all project members and to continuously motivate them to add precise effort data into the respective sheet. For the interviews, an interview guideline was used by the interviewer and the interview was accompanied by a walkthrough of relevant project documentation to assure the completeness of the interview results.

The recommendations that were provided as a result of the producibility analysis have not been applied to the project. They have been discussed during the project retrospective and the project team consisting of supervisors and students estimated, if the recommendations would have solved the problems experienced in the project with the critical elements. Hence, the conclusion that the recommendations are able to solve the experienced problems is based on expert opinion (in the case of the supervisors), but also on the opinion of students that are less experienced in general.

Internal Validity

The group of students was well representing a typical team of master students. As the curriculum of the master students from the University of Kaiserslautern and from the international students are aligned to each other, we can assume that they have a similar knowledge on Software Engineering foundations. Nevertheless, we conducted short tutorials on the main Software Engineering topics relevant for the project like requirements engineering, UI and interaction design, architectural design, and quality assurance in the beginning of the project.

Some of the students started into the project with more programming experience than others, but none of them had any experience in iOS development, which was the programming language used by most of the students in the team. We taught them development for iOS in another tutorial in the beginning of the project. Java and C# have been used in the project in addition. Each student had previous experience in at least one of these two programming languages.

The students were not bothered in any form during the project by the experiment. Hence, we do not expect maturation effects. They knew that we collect effort data anyway to measure the overall project performance, which is also interesting to them. Furthermore, they knew that the effort data are not used to measure their individual performance or to determine their grade in the end. Only not reporting effort data would have an effect on their grade as this would mean that they violate the project rules.

We were able to motivate them for the project retrospective as this was a good preparation for the final exam with the responsible professor. In the exam, they should be able to report on their role in the project, on problems and lessons learned, which we also discussed in the project retrospective.

As mentioned above, we provided them with an effort collection sheet and performed the interviews in the retrospective in a structured way to prevent threats caused by instrumentation.

Construct Validity

A thread to construct validity is the fact that the method owner himself performed the producibility analysis in the case study. The identification of the critical elements based on the quality model of producibility is only slightly affected by this. The producibility metrics are objective and additional critical elements have been identified by means of checklists. The method owner stuck to such checklists as far as possible.

The case study investigates the effect of a producibility analysis on a project that has been completely conducted without performing a producibility analysis. This setting is valid as there is no similar approach than the producibility analysis method described in this thesis that would allow a reasonable comparison.

The input documents provided to the producibility analysis method have been of appropriate quality. It can be assumed that the supervisors of the project performed project planning according to the state of the art based on their existing experience in industry and research projects. The students designed an architecture according to a state of the art architectural design approach. To assure the quality of the architecture designed by the students, one supervisor was actively involved into the design process and another one performed the quality assurance of the results.

The students did not know that there is a case study going on unless they have participated in the interviews and were confronted with the results of the producibility analysis.

External Validity

The case study is based on an industrial problem in the domain of mobile assistant systems. Projects developing similar mobile systems with similar architectures can directly benefit from the results of the case study. As the project covered certain typical problems of distributed information systems, there is also a certain benefit for distributed information system in general.

The size of the overall system is still small, i.e. the number of architectural elements and production planning elements. There was no distributed production team, as it is common in many large projects today. Hence, the scalability of the producibility analysis method to large, distributed projects has not been covered in the case study and the applicability to larger problems remains unclear. In general, we expect an even larger effect of a producibility analysis in larger systems, as the dependencies between architectural elements and production planning elements get more complex and cause even more problems than in smaller systems.

The project was conducted by a team of master students. We have to consider this as a threat to validity as they are close to finishing their studies but less experienced as Software Engineers from industry. Hence, certain doubts remain if the problems that have been detected in the project would also have occurred in a purely industrial setting. We addressed this threat by providing the students all the support they need. The students were located in the same building than the supervisors and the supervisors regularly were present in the team room of the students to be able to address evolving problems as soon as possible.

6.3 Summary and Future Validation Steps

Summary

The work on thesis was accompanied by a series of industrial and applied research projects that contributed to the problem statement, the solution ideas, and to initial validation of the ideas. Two projects have been specifically relevant.

In the project "Virtual Office of the Future" (VOF) conducted with the Japanese manufacturer of Multi-Functional Office Peripherals (MFP) the idea of software production came up and has been initially validated. In the VOF project, we identified the need to produce workflows and variants of workflows on a large scale as part of office environments integrating MFPs but also various office services. Together with Ricoh, we were able to define a reference architecture for office environments complemented by production processes tailored to the reference architecture. The production process describes, how workflows of a customer

can be realized in the context of the reference architecture and how existing and new office devices and services can be easily integrated. Ricoh expects to be able to deliver customized office environments with short time to market based on the ideas of software production. The project provided valuable input to the quality model of producibility.

In a series of industry projects with a large airline manufacturer, the idea of software production could be adopted in the context of the migration of a large workflow-based information system. The idea of migrating workflow by workflow can be supported with production processes and production. Unfortunately, the project was canceled on the customer's side before we could gather more results. But the project again showed the potential of the idea of software production and provided input to the definition of the quality model of producibility.

The VOF project, the projects in the airline management domain, and several other industrial projects provided valuable feedback on the idea of software production, we had the chance to adopt the ideas to industrial examples, and input for building the quality model of producibility has been gathered.

The producibility analysis method and the hypothesis stated in Section 1.5 have been evaluated by means of a case study conducted in an industrial context with a project team mainly consisting of master students. As described in detail in Section 6.2, a mobile configuration assistant has been developed in cooperation with John Deere. The case study delivered promising results with respect to all three hypothesis H1, H2, and H3. The producibility analysis method has been effective in the case study as it detected 91,6% of the critical elements, which is more than the 75% mentioned in H2. The results have also been almost completely correct as 95% of the identified critical elements have been really critical in the project which is beyond the 90% stated in H3. It has been detected, that 29% of time could have been saved in production iteration 1 if the problems related to the critical elements would have been solved and that also a significant amount of effort could have been saved for later phases of the project which eventually would have made it possible to complete the project as planned. Hence, initial empirical evidence that H1 can be accepted has been delivered.

However, a series of threats to validity must be considered in the context of the case study. The case study provides only one data point. Hence, the results cannot be considered to be highly significant. The producibility analysis method has been adopted by the method owner. The team mainly consisted of master students that are less experienced than Software Engineers in industry. The problem to be solved by the students was a real industrial problem, but relatively small.

Nevertheless, the case study provides indications that the expected effects of the producibility analysis method can be achieved in practice.

We expect an even larger effect of the producibility analysis method if applied to a larger scale industrial problem. The larger the problem is, the more dependencies between architectural and production planning elements occur that cannot easily be foreseen and lead to production problems.

Future Validation Activities

The last argument is a good starting point for discussing potential future validation activities.

The producibility analysis method should be adopted in a large-scale industrial case study. Unfortunately, this was not possible in the validation phase of this thesis as appropriate projects were not available at that time. In a large project, more architectural and production planning elements are involved and their relationships are expected to be more complex and can no longer be easily overseen. Hence, it gets more and more important to get support in identifying critical architectural and production planning elements. We also assume that in large scale project with a longer duration and more resources involved, the potential to save time and effort by improving the producibility is higher than in smaller projects.

Experiments to validate the hypothesis H1, H2, and H3 are not considered with high priority. Experiments typically refer to small examples and as argued above the producibility analysis method should be adopted to large industrial systems to unfold its full potential. Experiments could be conducted to validate specific aspects or parts of the producibility analysis method, for instance, to validate the appropriateness of the producibility views to detect production problems or to evaluate the usability of the provided checklists. Such experiments could provide useful input to improve certain aspects of the method and in the end contribute to improve the overall performance of the method.

Controlled experiments should be considered to validate the quality model of producibility and test the validity of the producibility metrics. Various experiment settings are possible depending on the concrete goals regarding the validation of the quality model of producibility. One goal could be, for instance, to validate if certain producibility metrics correlate with the appearance of certain production problems. Critical values of metrics regarding the alignment of architecture and production schedule, for instance, are supposed to correlate to production problems like delays. Variants of an architecture and a corresponding production plan could be derived that cause different values regarding metrics characterizing the alignment of architecture and production plan. The variants could then be produced by different groups of participants of the experiment and production problems could be tracked. The basic hypothesis of such an experiment would be that variants of architecture and production plan with critical values of the respective producibility metrics cause more or more severe production problems than variants with less critical values.

7 Summary and Future Work

This chapter summarizes the contributions of this thesis and gives an outlook on future work.

7.1 Summary of Contributions

Definition of Software Production

This thesis defined software production as a process "creating and assembling the architectural elements defined in the software architecture according to a software production plan" (see definition of software production in Section 3.1.). Software production is inspired by the manufacturing of hard goods, where product designers and production planners thoroughly plan production based on the product design and try to identify potential production problems up-front, i.e., before production starts. Therefore, the definition of software production in Section 3.1 claims, that "software architecture and software production plan have been aligned to each other". Alignment means in this case, that the relationships between architectural elements and production planning elements are explicitly considered and eventual misalignments are proactively resolved. If the same architectural element is planned to be modified by different resources according, for instance, there is a certain risk that conflicts and unexpected side effects arise that lead to production problems like delays or effort overhead. If adopted consequently, the idea of software production leads to a very product-oriented thinking, which we experienced to be very helpful in several industrial projects.

Meta-Model of Software Production

The definition of software production has been formalized in a meta-model of software production (see Section 3.4). As a prerequisite, meta-models of software architecture and software project plans have been derived as part of this thesis (see Chapter 2). The meta-model of software production integrates these meta-models of software architecture (see Section 2.1) and software project plans (see Section 2.2) by introducing relationships between conceptual elements of software architecture like architectural elements and conceptual elements of project plans like work activities, iterations, or resources. By relating architectural elements with work activities, iterations, and resources, the relationship of software architecture and project or production plans becomes as concrete as required to define the alignment of architecture and project or production plans in a measurable way, which is done in the quality model of producibility (see Chapter 4).

Definition of Producibility

Producibility is introduced in this thesis as a quality attribute of a system characterizing the alignment of its architecture and the production plan set-up to produce it. In Section 4.1, producibility is defined as "the degree of alignment of a system's architecture with the production plan". Producibility is considered on three dimensions, namely alignment of architecture and production work breakdown structure, architecture and production schedule, and architecture and resource assignments. The rationale for having these three dimensions is that according to the state of the practice and the state of the art software project planning mainly deals with project scope manifested in a work breakdown structure, with project schedule, and resource assignments. Hence, the alignment of architecture and production plans is covered comprehensively in the definition of producibility.

Quality Model of Producibility

The quality model of producibility defines producibility in a measurable form by providing metrics characterizing the alignment of architecture and production plan. Producibility metrics are introduced for all three dimensions of the alignment of architecture and production plan mentioned before. Thereby, we distinguish metrics that are rather relevant from the architect's point of view (see Section 4.2.1, 4.3.1, and 4.4.1) and the production planner's point of view (see Section 4.2.2, 4.3.2, and 4.4.2).

The metrics from an architect's point of view primarily are supposed to help architects in deciding if the architecture could be changed to increase producibility. Production planners are supported by the metrics for the production planner's point of view in considering changes of the production plan to eventually improve the producibility. Examples for architecture related metrics are the number of production work activities producing an architectural element, the number of production iterations involving a certain architectural element, the duration for producing an architectural element, or the number of resources producing an architectural element. Examples for production plan related metrics are the number of architectural elements involved in a certain production iteration, the degree of coverage of the overall system by one iteration, the overlapping between two sequential iterations in terms of architectural elements, or the coupling between resources caused by the coupling between architectural elements.

The producibility metrics have been derived systematically from the meta-model of software production and consequently cover all relations between architectural and production planning elements specified there. Metrics of different producibility dimensions are largely independent of each other, i.e., they cover different aspects of producibility and typically do not correlate. A high value of Coupling(PI), for instance, does not imply a high value of Coupling (Res).

Each producibility metric is complemented by a threshold, that indicates if a value of the producibility metric must be considered critical. Howev-

er, based on our experience the values and the respective thresholds always need to be considered in the respective context. Hence, besides introducing producibility metrics, the quality model of producibility contains a set of context factors that are supposed to be considered while interpreting the values of producibility metrics as they have the potential to compensate values classified as critical before.

It is important to mention, that the producibility of a system cannot be characterized by one single metric. It depends on the context of a project and the specific production requirements that might exist which metrics are specifically relevant and how conclusions are drawn. We introduced the concept of producibility scenarios in this thesis. They are similar to architectural scenarios as they are often used to refine quality requirements in the architecture. They can be used to specify production requirements and the overall producibility can be evaluated relative to such producibility scenarios.

Producibility Views

Producibility views have been defined to model the relationships of architectural and production planning elements in concrete projects and determine the producibility metrics of the quality model of producibility based on such views. Three producibility views have been defined, one for each dimension of producibility (see Section 5.2.2). The production work activity view is used to model relationships of architectural elements and production work activities. The production iteration view shows how architectural elements are related to production iterations. The resource assignment view relates architectural elements and resources. Producibility views should be part of the architecture and the production plan documentation as they are the artifacts manifesting the relationship between architecture and production plans and should be considered by architects as well as production planners.

Identification of Critical Elements

The thesis describes an algorithm that identifies critical architectural and production planning elements. The algorithm takes as input the information modeled in producibility views and adds critical architectural elements, critical production work activities, critical production iterations, and critical resources to respective lists of critical elements that form the output of the algorithm. To identify elements as critical, the algorithm determines values for the producibility metrics specified in the quality model of producibility and compares such values to the thresholds proposed for each metric. If the value for a certain element indicates criticality, the element is placed on the respective output lists, i.e., the list of critical architectural elements, the list of critical production work activities, the list of critical production iterations, or the list of critical resources.

Initial Prototypical Tool Support

Initial prototypical tool support has been implemented for the identification of critical elements. The producibility views can be modeled by means of the industrial modeling tool Enterprise Architect that has been extended for this purpose. Based on the modeled producibility views, most of the producibility metrics being part of the quality model of producibility can be determined automatically.

The producibility views, the identification of the critical elements based on the quality model of producibility, and the initial tool support are the core technical contributions of this thesis.

Producibility Analysis Method

The producibility analysis method is the methodological contribution of this thesis (see Chapter 5). Based on an existing software architecture and a software production plan, the producibility analysis method identifies critical architectural and production planning elements based on the algorithm mentioned above and guides the method users in deriving recommendations on how to prevent production problems like delays, and effort overhead. The producibility analysis method can be focused on certain aspects if required by using producibility scenarios that describe production requirements from different stakeholders.

In the preparation phase of the method (see Section 5.2), producibility scenarios are elicited from stakeholders like architects, production planners, customers, or developers. A template for documenting producibility scenarios is provided. A further step in the preparation phase is the modeling of the producibility views required to perform the identification of critical elements. The producibility views can be modeled by means of the tool Enterprise Architect that has been extended with modeling capabilities for producibility views as mentioned above.

In the execution phase of the producibility analysis method (see Section 5.3), critical architectural and production planning elements are identified based on the algorithm mentioned before and based on expert judgment. The algorithm has certain limitations. Sometimes, architectural elements, for instance, turn out to be critical although no metrics showed critical values. Architectural elements could have a high inherent complexity, for instance, which cannot be detected by the algorithm. Hence, architects and production planners are provided with examples of critical elements not detected by the algorithm that help them in identifying additional critical elements, which is the initial step of the consolidation phase.

In the consolidation phase of the method (see Section 5.4), architects and production planners first consolidate the critical elements that are reported to them from the execution phase. They take decisions if such elements appear really critical to them in the respective project context. Checklists containing questions derived from the context factors of producibility (see Section 4.5) that help them to take their decisions support this step. After the consolidated lists of critical elements exist, architects

and production planners derive appropriate recommendations on how to prevent problems that can potentially be caused by the critical elements. They are supported in this step by lists of typical options for recommendations. Recommendations are derived jointly by architects and production planners. This is required, because preventing production problems might be possible in various ways, i.e., for instance, by changing the architecture or by changing the production plan or a combination of both. Therefore, decisions on recommendations need to be taken jointly from an overall project perspective.

Validation

The producibility analysis method has been initially validated in this thesis (see Chapter 6). In an industrial case study conducted with a team of master students of the University of Kaiserslautern (see Section 6.2), initial empirical evidence for the research hypotheses stated in Section 1.5 has been gathered. In the case study, 91,67% percent of critical elements have been identified by the method and a correctness of 95% has been achieved. It has been estimated, that delays caused by the critical elements could have been prevented which would have reduced the time for the initial iteration of the project by 29%. We also found indications that the effort spent on certain production iterations in the project could have been reduced by means of the producibility analysis method which would potentially have enabled the project to realize the complete functionality that was initially planned. Actually, the project without using the results of the producibility analysis was successful overall, but did not realize the functionality that was originally planned due to delays and effort overhead caused by the identified critical elements.

7.2 Outlook on Future Work

Agile Software production

The combination of the ideas of software production and agile methods to an approach that could be called agile software production seems to be very promising.

Software production is essentially product-oriented, as it puts the architecture in the center of the production process. Per definition, it takes care that each production work activity contributes to the product as certain architectural elements are produced, i.e., either newly created or modified. But it can be the case, that certain production iterations do not significantly increase the business value of the current version of the product. If a production iteration produces and integrates certain infrastructure architectural elements but no architectural elements realizing business functionality using such infrastructure components, the business value of such a production iteration might be low also it makes sense to have such an iteration from a producibility point of view.

According to the Agile Manifesto [AM01], agile methods aim at "satisfying the customer through early and continuous delivery of valuable

software". Agile methods are iterative and incremental and aim at providing new features to a customer in each iteration or release. Hence, agile methods are also essentially product-oriented. Agile methods disapprove huge up-front investments into software architecture. They promote a more evolutionary approach to software architecture and propose refactoring to solve eventual design issues. They justify their approach by continuous changes of requirements that bear the risk that up-front investments never pay off. The risk of agile methods is that a feature required by the customer at a certain point in time somehow breaks the architecture. Breaking the architecture could mean in this case, that the feature requires the majority of architectural elements of the system to be changed.

Software production and agile methods could be complementary and potential weaknesses of one or the other approach eventually could be compensated. The alignment of software architecture and production plan takes care that realizing a certain feature at a certain point in time does not break the architecture in the sense that the majority of architectural elements would need to be changed. Such a situation could be detected by, for instance, a producibility analysis. This would require architectural design up-front to a certain degree to enable a "lightweight" producibility analysis, for instance. Agile methods could force software production to more explicitly make sure that each production iteration provides a certain business value and a positive return on investment can be achieved potentially earlier.

Future research in the direction of agile software production is required and seems to be promising. A concrete idea could be, for instance, to investigate how architecture could be used in the iteration and release planning performed in agile projects to influence the scope of upcoming iterations and releases.

Tailoring Quality Model of Producibility

The quality model of producibility defines producibility metrics and related optimal values. The algorithm to detect critical elements uses such optimal values and classifies elements as critical as soon as the optimal value is not achieved. This is a rather pessimistic strategy, as each deviation from the optimal value is supposed to cause production problems. But there are cases where already the smallest deviation causes a production problem. As soon as an architectural elements is modified in parallel by two resources, conflicts and related problems can occur, but not necessarily need to. In future work on the producibility model, it should be investigated in which situations certain deviations of the optimal value are acceptable and the respective elements do not need to be classified as critical. This investigation should consider context factors and should be based on empirical data collected in software production projects.

Modeling of Context Factors in Producibility Views

Three producibility views have been introduced in this thesis. So far, the producibility views contain all information relevant to determine the producibility metrics and identify critical elements in the execution phase of the producibility analysis method. So far, the context factors being part of the quality model of producibility are not modeled in the producibility views and therefore cannot be used for the identification of critical elements. In the future, context factors should be considered to be modeled in the producibility views. Some context factors can be modeled as properties of the elements used in the producibility views. Technologies could be modeled as properties of architectural elements. For production iteration, the respective technology mix could then be automatically determined and visualized.

Context information modeled in the producibility views could be used in the algorithm to identify critical elements. The context factors could extend the automatic identification of critical elements and reduce the manual effort required to identify additional critical elements based on context factors.

Additional Producibility Views

Additional producibility views can be imagined. One promising example could be an architecture evolution view consisting of a sequence of structural views of the architecture. It is similar to a manual to assemble a piece of furniture. The architecture evolution view shows a structural view of the architecture showing a snapshot of the system at the end of each production iteration. The structural views are ordered according to the sequence of the production iterations. The delta to the structural view of the previous production iteration can be visualized in each structural view. The architecture evolution view visualizes the progress made in each production iteration in terms of architectural elements. It provides a view of the system that is reduced to the scope planned for the end of the current production iteration, leaves out future extensions, and simplifies the current view on the system for the production team. It helps to visually detect complex production iterations and potential production problems.

Enhancements of the Producibility Analysis Method

Various enhancements of the producibility analysis method can be imagined that could further improve the methods effectiveness and efficiency.

In the preparation phase, the modeling of producibility views could be better supported. If the production plan refers, for instance, to features but traceability information would be modeled, this could be used as a basis to at least partially generate producibility views. This would reduce the effort of modeling producibility views.

For the execution phase, an extended version of the algorithm to identify critical elements could be developed that considers context factors. The assumption for this extension would be that context information is modeled in the producibility views, as mentioned above. This would increase

the degree of automation in detecting critical architectural elements. The quality of the results of the algorithm could be further improved by using tailored quality models of producibility as mentioned above.

With respect to the consolidation phase, the derivation of recommendations could be better supported in the future. Currently, lists of recommendations showing the general options available guide the method users. The selection of one of the options today is completely human based. In the future, it should be considered to semi-automate the selection of recommendations. Certain recommendations could be preselected automatically based on certain rules derived from the quality model of producibility maybe under further consideration of empirical data. A human could then take the final decision.

The approach could be easily extended to be able to deal with customization and deployment projects of, for instance, standard software solutions. Deplyoment units have already been specified as a type of architectural elements in Chapter 2.1. Deployment views could be added to the set of producibility views, and customization and deployment work activities could be added to the production plan. Then, the method specified in Chapter 5 in general could be adopted to such a customization and deployment project.

Design for Producibility

The producibility analysis method is an analytic approach to improve the producibility of a system. In the future, constructive approaches should be considered that, for instance, support architects in designing for producibility. This is an approach that again can be adopted from other engineering disciplines that consider the whole product lifecycle already during design, from manufacturability [Bra98] to recyclability [Fik09]. This design process is even tool-supported by Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) tools.

Future Validation

As already mentioned in Section 6.3, further validation of the producibility analysis method is required to gather empirical evidence that the research hypotheses of this thesis can be accepted. The overall producibility analysis method should validated in a large industrial case study. Unfortunately, this was not possible so far during the course of this thesis. We expect the full power of the producibility analysis method to be unrolled in larger projects where the number of relationships between architectural elements and production planning elements and their complexity further increases. Such industrial case studies would be preferred over experiments. Nevertheless, experiments are the right approach to validate specific aspects of the producibility analysis method like the applicability of producibility views or the checklists provided to adopt the context factors of producibility. Hence, by means of experiments, specific aspects of the producibility analysis method could be improved and the overall effectiveness, as it has been mentioned in hypothesis H1 of this thesis, could be enhanced. Furthermore, experiments should be con-

ducted to validate the quality model of producibility and its metrics regarding their explanatory power.

7.3 Concluding Remarks

This thesis enters a new field of Software Engineering research by systematically investigating the relationship of software architecture and software project or production plans. It introduces producibility as a new quality attribute characterizing the alignment of software architecture and project or production plans. In several projects, we experienced the huge potential of systematically addressing producibility to reduce delays, effort overhead, and quality issues of the final product. The contributions of this thesis enable practitioners to exploit this potential. However, as this is the first dissertation thesis in this direction, it should be seen as a starting point for further research.

References

[ABB+01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wust, J. Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, 2001.

[Ada10] S. Adam. *Improving SPL-based Information System Development Through Tailored Requirements Processes*. 18th International Requirements Engineering Conference, Doctoral Symposium, Sydney, 2010.

[AJM06] S. T. Acuna, N. Juristo, A.M. Moreno. *Emphasizing Human Capabilities in Software Development*, IEEE Software, Volume 23, Issue 2, pp. 94-101, 2006.

[AM01] Agile Manifesto
http://agilemanifesto.org/
- last visited 31.03.2011 –

[Amb02] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley, 2002.

[And] Android Development Platform
http://www.android.com/
- last visited 31.03.2011 –

[BA04] K. Beck, C. Andres. *Extreme Programming Explained: Embrace Change*. Pearson, 2nd Edition, 2004.

[BAB+00] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, B. Steece. *Software Cost Estimation with Cocomo II*, Prentice Hall International, 2000.

[Bak72] F. T. Baker. *Chief programmer team management of production programming*. IBM Systems Journal, Volume: 11 Issue:1, pp. 56 – 73, 1972.

[Bas93] V. Basili. *The Experimental Paradigm in Software Engineering*, Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions, 1993.

[BCR94] V.R. Basili, C. Caldiera and H.D. Rombach. *Experience Factory*, Encyclopediaof Software Engineering, 1, J. J. Marciniak, ed., John Wiley & Sons, 1994, pp. 469-476.

[BCR94b] V. Basili, G. Caldiera and H.D. Rombach, *The Goal Question Metric Approach*, Encyclopedia of Software Engineering, John Wiley & Sons, 1994, pp. 528–532.

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*, Second Edition, SEI Series in Software Engineering, Addison-Wesley, 2003.

[BA04] K. Beck, C, Andres. *Extreme Programming Explained: Embrace Change* (2nd Edition). Addison-Wesley Professional, 2004.

[BFK+99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud. *PuLSE: A Methodology to Develop Software Product Lines*, Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), pp. 122–131, 1999.

[BJN+06] M. Broy, M. Jarke, M. Nagel, D. Rombach. *Manifest: Strategische Bedeutung des Software Engineering in Deutschland*. Informatik-Spektrum, Volume 29, Number 3, pp. 210-221, 2006.

[BKR09] S. Becker, H. Koziolek, R. Reussner. The Palladio component model for model-driven performance prediction. Journal of Systems and Software, Volume 82, Issue 1, January 2009, pp. 3-22.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[Bra98] J. Bralla. *Design for Manufacturability Handbook.* McGraw-Hill Professional, 2nd edition, 1998.

[Cam07] G. Campbell. Software-Intensive Systems Producibility: A Vision and Roadmap (v 0.1). Technical Note, CMU/SEI-2007-TN-017, 2007.
http://www.sei.cmu.edu/library/abstracts/reports/07tn017.cfm
- last visited 31.03.2011 –

[Car08] R. Carbon. *Improving the Production Capability of Product Line Organizations*. SPLC 2008, 12th International Software Product Line Conference. Proceedings. Second Volume : Limerick, Ireland, 8-12 Sept 2008.

[CAU+09] R. Carbon, S. Adam, T. Uchida. *Towards a product line approach for office devices - facilitating customization of office devices at Ricoh Co Ltd*. In Proceedings of the 13th International Software Product Line Conference, SPLC 2009. Vol.1, pp. 151-160, San Francisco, USA, 2009.

[CBB+03] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures. Views and Beyond*, SEI Series in Software Engineering, Addison-Wesley, 2003.

[CJK+08] R. Carbon, G. Johann, T. Keuler, D. Muthig, M. Naab, S. Zilch. *Mobility in the virtual office: a document-centric workflow approach.* In Proceedings of the 1st international workshop on Software architectures and mobility, SAM '08,Leipzig, Germany, 2008.

[CJM+08] R. Carbon, G. Johann, D. Muthig, M. Naab. *A method for collaborative development of systems of systems in the office domain*. In Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference, pp. 339-345, Munich, Germany, 2008.

[CKK01] P. Clements, R. Kazman, M. Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley Professional, 2001.

[CLM+06] R. Carbon, M. Lindvall, D. Muthig, P. Costa. *Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design.* In Proceedings of the 1st International Workshop on Agile Product Line Engineering, APLE06, Baltimore, USA, 2006.

[CM02] G. Chastek, J. D. McGregor. *Guidelines for Developing a Product Line Production Plan*. Technical Report, CMU/SEI-2002-TR-006, 2002.

[CN02] P. Clements, L. Northrop. *Software Product Lines – Practices and Patterns*, SEI Series in Software Engineering, Addison Wesley, 2002.

[Coh09] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009.

[Con68] M. E. Conway. *How do Committees Invent?* Datamation 14 (5): 28–31, http://www.melconway.com/research/committees.html - last visited 31.03.2011 –

[Con93] L.L. Constantine. *Work Organization: Paradigms for Project Management and Organization*. Communications of the ACM, 36(10), pp. 34-43, 1993.

[DFK98] J. DeBaud, O. Flege, P. Knauber. *PuLSE-DSSA—a method for the development of software reference architectures,* in Proceedings of the third international workshop on Software architecture (ISAW '98), pp. 25-28, 1998.

[EA11] Sparx Enterprise Architect http://www.sparxsystems.de/ - last visited 31.03.2011 –

[Fair09] R.E. Fairley. *Managing and Leading Software Projects*. IEEE Computer Society. John Wiley and Sons Inc., 2009.

[Fik09] J. Fiskel. *Design for Environment, Second Edition: A Guide to Sustainable Product Development: Eco-Efficient Product Development*. McGraw-Hill Professional, 2nd edition. 2009.

[GHJ+94] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional,1994.

[GRD+97] S. Gupta, W. Regli, D. Das, D. Nau, Automated Manufacturability Analysis: A Survey, Research in Engineering Design, (1997) 9, pp. 168-190, Springer London, 1997.

[GSC+04] J. Greenfield, K. Short, S. Cook, S. Kent, J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[Hen95] B. Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity (Object-Oriented Series)*. Addison Wesley, 1995.

[HND99] C. Hofmeister, R. Nord, S. Dilip. *Applied Software Architecture*, Addison-Wesley Object Technology Series, 1999.

[HOF11] The Produt Line Hall of Fame
http://www.splc.net/fame.html
- last visited 31.03.2011 –

[Hun06] J. Hunt. *Agile Software Construction*. Springer London, 2006.

[IEEE98] IEEE Standard 1058-1998 for Software Project Management Plans
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=741937
- last visited 31.03.2011 –

[iOS] iOS Development Platform
http://developer.apple.com/devcenter/ios/index.action
- last visited 31.03.2011 –

[ISO01] ISO/IEC 9126-1:2001, Software engineering -- Product quality -- Part 1: Quality model
http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749
- last visited 31.03.2011 –

[ISO07] ISO/IEC 42010:2007, Recommended Practice for Architectural Description of Software-intensive Systems
http://www.iso.org/iso/catalogue_detail.htm?csnumber=45991
- last visited 31.03.2011 -

[ISO09] ISO/IEC 20926:2009, IFPUG functional size measurement method 2009
http://www.iso.org/iso/catalogue_detail.htm?csnumber=51717
- last visited 31.03.2011 –

[Jal10] P. Jalote. *A Concise Introduction to Software Engineering*. Springer London, 2010.

[Java11] Java EE Development Platform
http://download.oracle.com/javaee/
- last visited 31.03.2011 –

[JSON] Java Script Object Notation
http://www.json.org/
- last visited 31.03.2011 –

[JRL00] M. Jazayeri, A. Ran, F. van der Linden. *Software Architecture for Product Families – Principles and Practice*. Addison Wesley, Pearson Education, 2000.

[KAB+96] R. Kazman, G. Abowd, L. Bass, P. Clements. *Scenario-based analysis of software architecture*. IEEE Software 13 (6), pp. 47 – 55, 1996.

[Kel61] J. Kelley. *Critical Path Planning and Scheduling: Mathematical Basis*. Operations Research, Vol. 9, No. 3, pp. 296-320, May-June, 1961.

[KKC00] R. Kazman, M. Klein, P. Clements. *ATAM: Method for Architecture Evaluation*. Technical Report, CMU/SEI-2000-TR-004, ESC-TR-2000-004, 2000.

[KMH+08] J. Knodel, D. Muthig, U. Haury, G. Meier. *Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry.* In Proceedings of the 12th European Conference on Software Maintenance and Reengineering. CSMR, pp. 43-52, 2008.

[Kno09] J. Knodel. *From architecture to source code - how to ensure architecture compliance in the implemented system.* Softwaretechnik-Trends 29 (2009), No.2, pp.13-14, 2009.

[Kru95] P. Kruchten. *Architectural Blueprints — The "4+1" View Model of Software Architecture.* IEEE Software 12 (6), pp. 42-50, 1995.

[Krue01] C. W. Krueger. *Easing the Transition to Software Mass Customization*. In: Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE '01), Springer, London, 2002.

[Krue02] C. W. Krueger. Variation Management for Software Production Lines. In: Proceedings of the Second International Conference on Software Product Lines (SPLC 2), Springer, London, 2002.

[Kru03] P. Kruchten. *The Rational Unified Process: an Introduction*. Addison-Wesley Professional, 2003.

[Kru06] C.W. Krueger. *New methods in software product line development* .In Proceedings of the 10th Software Product Line Conference, SPLC 2006, pp. 95 – 99, Baltimore, USA, 2006.

[Lar10] C. Larman. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley Professional, 2010.

[LM10] A. Lamersdorf, J. Münch. *Model-based Task Allocation in Distributed Software Development.* In Proceedings of the Fourth International Conference on Software Engineering Approaches For Offshore and Outsourced Development, SEAFOOD 2010, pp. 37-53, St. Petersburg, 2010.

[LT75] H. A. Linstone, M. Turoff. *The Delphi Method: Techniques and Applications*, Reading, Mass.: Addison-Wesley,1975.

[MC08] J. D. McGregor, G. Chastek. *Production Planning in a Software Product Line Organization*. In Proceedings of the 12th International Software Product Line Conference, Limerick, Ireland, 2008.

[McG04] J. D. McGregor. *Product Production.* In Journal of Object Technology, Vol. 3, No. 10, November-December 2004, pp. 89-98. http://www.jot.fm/issues/issue_2004_11/column7 - last visited 31.03.2011 –

[McG05] J. D. McGregor. *Preparing for Automated Derivation of Products in a Software Product Line.* Technical Report, CMU/SEI-2005-TR-017, 2005. http://www.sei.cmu.edu/library/abstracts/reports/05tr017.cfm - last visited 31.03.2011 –

[MSNet11] Microsoft .Net Platform
http://www.microsoft.com/net/
- last visited 31.03.2011 –

[MSProj10] Microsoft Project
http://www.microsoft.com/project/
- last visited 31.03.2011 –

[NC07] L.M. Northrop, P.C. Clements. *A Framework for Software Product Line Practice,* Version 5.0, Software Engineering Institute, 2007.

[NRC10] National Research Council, Committee for Advancing Software-Intensive Systems Producibility. Critical Code: Software Producibility for Defense. The National Academies Press, USA, 2010.
http://www.nap.edu/catalog/12979.html
- last visited 31.03.2011 –

[OB88] T. Ohno, N, Bodek. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[OpenGr09] The Open Group SOA Reference Architecture
http://www.opengroup.org/projects/soa-ref-arch/
- last visited 31.03.2011 –

[OpenUp] Open Unified Process
http://epf.eclipse.org/wikis/openup/
- last visited 31.03.2011 –

[Pau02] D. Paulish, *Architecture-Centric Software Project Management: A Practical Guide*, Addison-Wesley Professional, 2002.

[Perry98] D.E. Perry. Generic Architecture Descriptions for Product Lines. Development and Evolution of Software Architectures for Product Families, LNCS, Volume 1429/1998, pp. 51-56, 1998.

[PMBOK04] Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Third Edition*. Project Management Institute, 2004.

[PP03] M. Poppendieck, T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[RBH+07] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, M. Kuperberg. *The Palladio Component Model*. Technical Report, Chair for Software Design & Quality (SDQ), Karlsruhe Institute of Technology, Germany, 2007.

[RH08] R. Reussner, W. Hasselbring. Handbuch der Software-Architektur. 2nd Edition. dpunkt Verlag, 2008.

[RM05] G. Ruhe, J. Momoh. *Strategic Release Planning and Evaluation of Operational Feasibility*. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05).

[Rom03] H. D. Rombach. *A Process Platform for Experience-Based Software Development*. In Proceedings of the International Colloquium of the Sonderforschungsbereich 501, University of Kaiserslautern, pp. 47-57, 2003.

[Rom10] H. D. Rombach. *Lecture* "*Grundlagen des Software Engineering*", 2010, http://wwwagse.informatik.uni-kl.de/teaching/gse/ws2010 - last visited 31.03.2011 -

[RW05] N. Rozanski, E. Woods. *Software Systems Architecture – Working With Stakeholders Using Viewpoints and Perspectives.* Pearson Education Inc., 2005.

[SB01] K. Schwaber, M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[SGF+10] M. Svahnberg, T. Gorschek, R. Feldt, R. Torkar, S. B. Saleem, M. U. Shafique. *A systematic review on strategic release planning models*. JpurnalInformation and Software Technology, Volume 52 Issue 3, March, 2010.

[Sie04] J. Siedersleben. *Moderne Softwarearchitektur – Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 2004.

[SMC74] W. Stevens, G. Myers, L. Constantine. *Structured Design*, IBM Systems Journal, 13 (2), 115-139, 1974.

[Som10] I. Sommerville. *Software Engineering.* 9th Edition, Addison Wesley, 2010.

[SQL] Structured Query Language http://en.wikipedia.org/wiki/SQL - last visited 31.03.2011 –

[Sta09] The Standish Group Chaos Report http://www.standishgroup.com/newsroom/chaos_2009.php - last visited 31.03.2011 –

[Szy02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming (Component Software Series)*, Addison-Wesley Longman, 2002.

[TMD10] R.N. Taylor, N. Medvidovic, E.M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. John Wiley and Sons Inc., 2010.

[UE95] K. Ulrich, S. Eppinger. *Product Design and Development*. McGraw-Hill, Inc., 1995.

[VModellXT] V-Modell XT http://www.v-modell-xt.de/ - last visited 31.03.2011 –

[WBB+06] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, W. Wood. *Attribute-Driven Design (ADD)* , Version 2.0, Technical Report Software Engineering Institute, CMU/SEI-2006-TR-023, 2006.

[Wiki11] Wikipedia Definition Trade-Off - last visited 31.03.2011 –

[WP7] Windows Phone 7 Development Platform http://www.microsoft.com/windowsphone/en-us/default.aspx - last visited 31.03.2011 –

[WRH+00] C. Wohlin, P. Runeson, M. Höst, M.Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering – An Introduction*, Kluwer Academic Publisher, 2000.

[W3C04] W3C Web Service Glossary
http://www.w3.org/TR/ws-gloss/
- last visited 31.03.2011 –

Appendix A: Producibility Metrics and Conditions

Producibility Metric	Condition checked
#PWA_producing(AE)	>1
#PWA_consuming(AE)	>0
#AE_produced_by(PWA)	>1
#AE_consumed_by(PWA)	>1
Coupling(PWA)	>0
#Shared_AE(PWA)	>0
#PI_producing(AE)	>1
ProductionDuration(AE)	>1
#AE_involved_in(PI)	> round($\lvert AE_{all} \rvert$ / $\lvert PI_{all} \rvert$)
Coupling (PI)	>0
#Shared_AE(PI)	>0
%Shared_AE (PI)	>0
%Completed_AE_after(PI)	< IterationNumber(PI) / $\lvert PI_{all} \rvert$
%Created_AE_after(PI)	< IterationNumber(PI) / $\lvert PI_{all} \rvert$
#Resources_working_on (AE)	>1
#AE_worked_on_by(Res)	> round($\lvert AE_{all} \rvert$ / $\lvert RES_{all} \rvert$)
Coupling(Res)	>0
#Shared_AE(Res)	>0

Appendix B: Algorithm to identify Critical Elements

The algorithm takes as an input one instance of the producibility views:

```
Production Work Activity View PWAView

Production Iteration View PIView

Resource Assignment View RAView
```

The output of the algorithm are lists of critical architectural and production planning elements:

```
List ListOfCriticalAE

List ListOfCriticalPI

List ListOfCriticalPWA

List ListOfCriticalRes
```

The processing of the production work activity view PWAView works as follows:

```
//Processing the Production Work Activity View

For each AE in PWAView

AE.CriticalDimensions = 0

AE.#PWA_producing_AE = #PWA_producing_AE(AE)

AE.#PWA_consuming_AE = #PWA_consuming_AE(AE)

If    AE.#PWA_producing_AE > 1

      or AE.#PWA_consuming_AE > 1

Then  AE.CriticalDimensions=1

ListOfCriticalAE.add(AE)

For each PWA in PWAView
```

```
PWA.#AE_produced_by_PWA = #AE_produced_by_PWA(PWA)

PWA.#AE_consumed_by_PWA = #AE_consumed_by_PWA(PWA)

PWA.Coupling = Coupling(PWA)

PWA.#Shared_AE_PWA = #Shared_AE_PWA(PWA)

PWA.%Shared_AE_PWA = %Shared_AE_PWA(PWA)

If    PWA.#AE_produced_by_PWA > 1

      or PWA.#AE_consumed_by_PWA > 0

      or PWA.Coupling > 0

      or PWA.#Shared_AE_PWA > 0

Then  ListOfCriticalPWA.add(PWA)
```

The processing of the production iteration view PIView works as follows:

```
//Processing the Production Iteration View

For each AE in PIView

AE.#PI_involving_AE = #PI_involving_AE(AE)

AE.PI_creating = PI_creating(AE)

AE.PI_finishing = PI_finishing(AE)

AE.ProductionDuration = AE.PI_finishing -
AE.PI.creating + 1

If    AE. #PI_involving_AE > 1

Then  AE.CriticalDimensions++

If !ListOfCriticalAE.contains(AE)

Then ListOfCriticalAE.add(AE)

For each PI in PIView

PI.#AE_involved_in =  #AE_involved_in(PI)

PI.%System_Coverage = %System_Coverage(PI)
```

```
PI.Coupling = Coupling(PI)

PI.#Shared_AE = #Shared_AE(PI1)

PI.%Shared_AE = %Shared_AE(PI)

PI.%Completed_AE = %Completed_AE(PI)

PI.%Created_AE = %Created_AE(PI)

If    PI.#AE_involved_in > |AE_all| / |PI_all|

      or PI.Coupling > 0

      or PI.Shared_AE > 0

      or PI.%Completed_AE < IterationNumber(PI) /
|PI_all|

      or PI.%Created_AE < IterationNumber(PI) /
|PI_all|

Then  ListOfCriticalPI.add(PI)
```

The processing of the resource assignment view works as follows:

```
//Processing the Resource Assignment View

For each AE in RAView

AE.#Resources_working_on   =   #Resources_working_on
(AE)

If    AE.#Resources_working_on > 1

Then  AE.CriticalDimensions++

If !ListOfCriticalAE.contains(AE)

Then ListOfCriticalAE.add(AE)

For each RES in RAView

RES.#AE_worked_on = #AE_worked_on(RES)

RES.Coupling = Coupling(RES)

RES.#Shared_AE = #Shared_AE(RES)
```

```
RES.%Shared_AE = %Shared_AE(RES)

If    RES.#AE_worked_on > |AE_all| / |Res_all|

      or RES.Coupling > 0

      or RES.#Shared_AE > 0

Then  ListOfCriticalRes.add(RES)
```

Appendix C: Checklists for Context Factors

Checklist for the Application of Context Factors to AE

General questions for each AE:

- Is the quality of the architecture documentation with respect to the AE high?
- Is a production work activity type, i.e. a guideline describing how to produce the AE available?
- Are certain development activity types supporting the production of the AE, for instance, continuous integration, regression testing, generation of parts of the AE, etc.?
- Are tools providing specific support for the production of the AE?
- Can the AE be built based on reuse and is a process how to reuse attached to the reusable artifacts?
- Are the resources providing and potentially adapting the reusable artifacts available when the AE is supposed to be produced?
- Is the available team experienced with the technologies used to realize the AE?
- Is the AE produced by internal resources and are they co-located?
- If the AE is produced by external resources or internal resources that are not co-located, is an appropriate communication infrastructure in place, and an infrastructure that facilitates the exchange of artifacts?
- Are contact persons for the AE under analysis and for all related AE in place that can help in solving issues?

#PWA_producing(AE)

- Is a certain order defined for performing the PWAs?
- Are the PWAs performed by the same team (or by co-located teams)?
- Is the internal design of the AE prepared for parallel work and/or incremental extension?
- Are integration and test processes defined for the AE?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?
- Does the tool infrastructure support parallel production well?

#PWA_consuming(AE)

- Are the PWAs consuming the AE produced later on?
- Are the consuming PWAs performed by co-located resources?
- If the resources producing consuming PWAs are not co-located, are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Do the involved resources know each other personally?
- Is the quality of the architecture documentation high, especially the documentation of the interfaces of the AE?

#PI_producing(AE)

- Are the same resources producing the AE throughout all iterations?
- Is the internal design of the AE prepared for incremental extension?
- Are integration and test processes defined for the AE?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?

#Resources_working_on(AE)

- Are the resources producing the AE co-located?

- Are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Do the involved resources know each other personally?
- Does each involved team have one single point of contact, for instance, a chief programmer?
- Do the resources work on parts of the AE separated in the design of the AE?
- Is one resource responsible for integration, final test, and deployment of the AE?

Checklist for the Application of Context Factors to PWA

General questions for each PWA:

- Are all AEs involved in the PWA well described in the architecture documentation?
- Are the involved AEs following well-known architectural styles, patterns, etc.?
- Are production work activity types available describing how to produce the involved AEs?
- Do development activity types specifically support the production of the involved AEs?
- Are tools available to specifically support the production of the involved AEs?
- Are the resources experienced and familiar with the used technologies and tools?
- Is sufficient effort and time planned for the PWA?

#AE_produced_by

- Is the PWA realizing a cross-cutting feature?
- If the PWA is realizing a cross-cutting feature, does the architecture prescribe sufficiently how the cross-cutting feature is supposed to be realized in each AE?
- Is the complexity of the AEs produced by the PWA low?
- Are only well known technologies used to produce the AEs?
- Are the involved AEs initially created by the PWA?
- Can some of the AEs be produced based on reuse?
- If AEs are produced based on reuse, are the reusable artifacts available in time and are contact persons "owning" the reusable artifacts available?
- Is the team size sufficient to perform the PWA?

#AE_consumed_by

- Are the AEs consumed by the PWA finished in earlier production iterations?
- Are the AEs consumed by the PWA produced by the same resource?
- Are the consumed PWAs mostly infrastructure AEs?
- Are the consumed PWA using the same technologies, especially to communicate with them?
- Are the consumed AEs well documented, especially their interfaces?
- Do the consumed AE and the communication mechanisms follow well-known architectural patterns?
- Are appropriate quality assurance activities performed for the consumed PWAs?

Coupling

- Are PWAs causing ingoing relations to the PWA assigned to later production iterations?
- Are PWAs referenced by the PWA assigned to previous iterations?
- Are related PWAs produced by the same resources?
- Does the PWA produce mostly infrastructure AEs?
- Are the related PWA respectively the AEs involved into them well-described in the architecture documentation, especially their interfaces?

#Shared_AE

- Are the resources producing the shared AEs co-located?
- If the resources producing shared AEs are not co-located, are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Is the internal design of the AE prepared for parallel work or incremental extension?
- Are integration and test processes defined for the AE?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?
- Does the tool infrastructure support parallel production well?

Checklist for the Application of Context Factors to PI

General Questions on PIs:

- Are all AEs involved in the PI well described in the architecture documentation?
- Are the involved AEs following well-known architectural styles, patterns, etc.?
- Are production work activity types available describing how to produce the involved AEs?
- Do development activity types specifically support the production of the involved AEs?
- Are tools available to specifically support the production of the involved AEs?
- Are the resources experienced and familiar with the used technologies and tools?
- Can a large number of AEs in the PI be built produced based on reuse?
- If AEs produced by the PI are based on reuse, are the reusable artifacts required available in time and are contact persons "owning" the reusable artifacts available?
- Is parallel work during the PI possible to a large degree?
- Is some buffer time planned in the end of the PI?

#AE_involved_in or %System_Coverage

- Is the PI realizing cross-cutting features?
- If the PI is realizing cross-cutting features, does the architecture prescribe sufficiently how the cross-cutting feature is supposed to be realized in each AE?
- Is the complexity of the AEs produced by the PI low?
- Are only well known technologies used to produce the AEs of the PI?
- Are the involved AEs initially created by the PI?
- Can some of the AEs be produced based on reuse?
- If AEs are produced based on reuse, are the reusable artifacts available in time and are contact persons "owning" the reusable artifacts available?

Coupling

- Are AEs referenced by the PI classified as uncritical?
- Is the previous PI classified as uncritical?
- Are the AEs referenced by the PI in previous PI well-described in the architecture documentation, especially their interfaces?
- Do previous PI contain buffer time at the end?

%Shared_AE

- Are the resources producing the shared AEs co-located?
- If the resources producing shared AEs are not co-located, are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Is the internal design of the shared AE prepared for incremental extension?
- Are integration and test processes defined for the AE, especially regression tests in this case?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?
- Does the tool infrastructure specifically support the production of the shared AEs?

Checklist for the Application of Context Factors to Res

General Questions on Res:

- Is the resource experienced and familiar with the used technologies and tools?
- Is the resource well connected to the outside via communication and data sharing infrastructures?
- Is the capacity of the resource large enough to deal with the assigned AEs and PWAs?
- Is the resource internally well-organized, i.e. are appropriate processes to make decisions in place?
- Is a single point of contact available from the outside?
- Is the resource experienced in project management?
- Can the resource work provide required capacities even if unforeseen events occur like illness?

#AE_worked_on_by

- Is the resource realizing cross-cutting features?
- If the Res is realizing cross-cutting features, does the architecture prescribe sufficiently how the cross-cutting feature is supposed to be realized in each AE?
- Is the complexity of the AEs worked on low?
- Are all AEs worked on produced by means of the same technology?
- Are all AEs worked on by the resource of architectural element types known by the team or even of one architectural element type?
- Are production work activity types available for the AEs worked on?
- Is specific tool support provided for the AEs worked on?
- Are the AEs clearly assigned internally in case of a team?
- Can some of the AEs be produced based on reuse?
- If AEs are produced based on reuse, are the reusable artifacts available in time and are contact persons "owning" the reusable artifacts available?

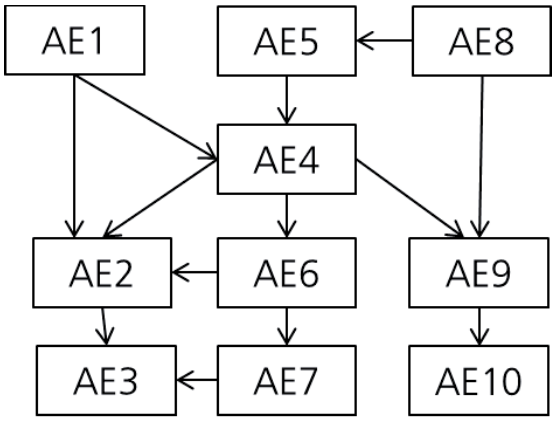
Coupling

- Are AEs referenced by the resource classified as uncritical?
- Are related resources not classified as critical?
- Are the AEs referenced by the Res well-described in the architecture documentation, especially their interfaces?

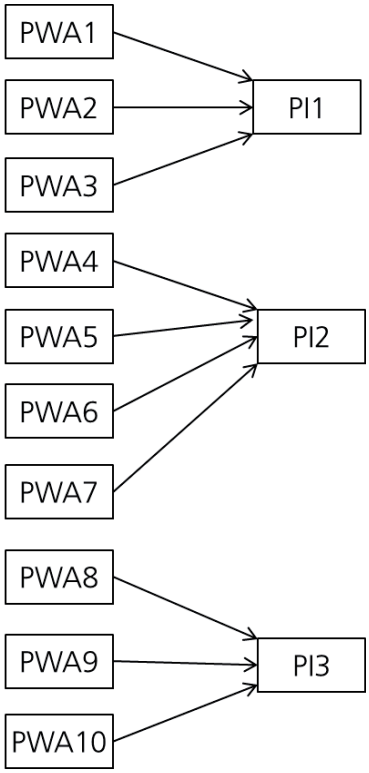
#Shared_AE

- Are the resources producing the shared AEs co-located?
- If the resources producing shared AEs are not co-located, are appropriate communication infrastructures and infrastructures to exchange artifacts established between the involved resources?
- Is the internal design of the shared AE prepared for incremental extension and parallel work?
- Are integration and test processes defined for the AE, especially regression tests in this case?
- Are deployment processes defined for the AE?
- Are coding guidelines defined for the AE?
- Does the tool infrastructure specifically support the production of the shared AEs?

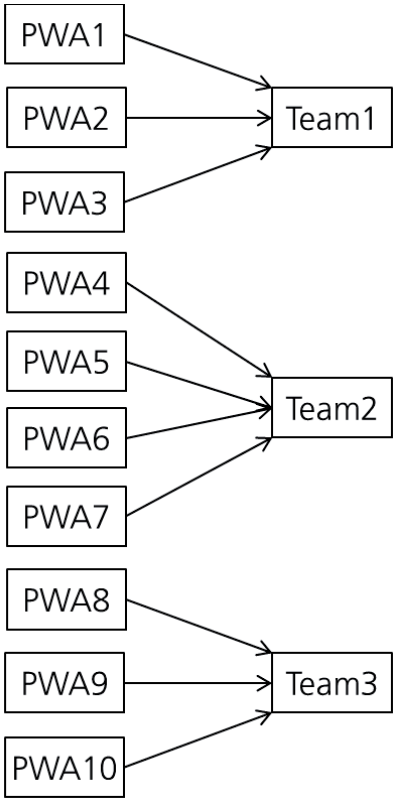
Appendix D: Method Example – Additional Materials



Appendix 1: Structural View



Appendix 2: Assignment of PWAs to PIs



Appendix 3: Resource Assignments

Appendix E: Case Study Results

The following tables show the values of all producibility metrics determined in the case study. The lists contain all elements, not only the critical ones.

AE	#PWA_producing	#PWA_consuming	#PI_producing	ProductionDuration	#Resources_working_on	CriticalDimensions
UI Manager	2	0	2	2	1	2
Control Unit	2	2	2	2	1	2
Model Manager	1	2	1	1	1	1
Distribution Manager	1	1	1	1	1	0
Domain Model Editor	1	0	1	1	1	0
Domain Model Base	1	2	1	1	1	1
Configuration Manager	1	2	1	1	1	1
Tractor Configuration Interface	1	1	1	1	1	0
Tractor Simulator	1	2	1	1	1	1
Tractor Simulator Configurator	1	0	1	1	1	0

PWA	#AE_produced_by	#AE_consumed_by	Coupling	#Shared_AE	%Shared_AE
Create UI Manager	1	1	2	1	100
Extend UI Manager	1	1	2	1	100
Create Control Unit	1	2	3	1	100
Extend Control Unit	1	2	3	1	100
Produce Model Manager	1	1	3	0	0
Produce Distribution Manager	1	1	2	0	0
Produce Domain Model Editor	1	1	1	0	0
Produce Domain Model Base	1	0	2	0	0
Produce Configuration Manager	1	1	3	0	0
Produce Tractor Configuration Interface	1	1	2	0	0
Produce Tractor Simulator	1	0	2	0	0
Produce Tractor Simulator Configurator	1	0	1	0	0

PI	#AE_involved_in	%System_Coverage	Coupling	#Shared_AE	%Shared_AE	%Completed_AE_after	%Created_AE_after
PI1	6	60	0	2	1/3	40	60
PI2	6	60	0	2	1/3	100	100

RES	#AE_worked_on_by	Coupling	#Shared_AE	%Shared_AE
Team1	1	1	0	0
Team2	3	2	0	0
Team3	6	1	0	0

Lebenslauf

Persönliche Daten

Name	Ralf Carbon
Anschrift	Marie-Juchacz-Str. 16 67663 Kaiserslautern
Geburtsdatum und -ort	13.05.1977 in Zweibrücken
Familienstand	Verheiratet, 1 Kind

Werdegang

1983 - 1987	Thomas-Mann Grundschule, Zweibrücken
1987 - 1996	Helmholtz-Gymnasium, Zweibrücken (Abitur)
1996 - 2002	Studium der Informatik, Universität Kaiserslautern (Diplom)
2002 - 2005	Wissenschaftlicher Mitarbeiter in der Arbeitsgruppe Software Engineering (AGSE), Technische Universität Kaiserslautern
seit Juli 2005	Wissenschaftlicher Mitarbeiter am Fraunhofer Institut für Experimentelles Software Engineering (IESE), Kaiserslautern

Kaiserslautern, den 16.11.2011

PhD Theses in Experimental Software Engineering

Volume 1 **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*

Volume 2 **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*

Volume 3 **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*

Volume 4 **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*

Volume 5 **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*

Volume 6 **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*

Volume 7 **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*

Volume 8 **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*

Volume 9 **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*

Volume 10 **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*

Volume 11 **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*

Volume 12 **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*

Volume 13 **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*

Volume 14 **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*

Volume 15 **Joachim Bayer** (2004), *View-Based Software Documentation*

Volume 16 **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

Volume 17 **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*

Volume 18 **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*

Volume 19 **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*

Volume 20 **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*

Volume 21 **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*

Volume 22 **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*

Volume 23 **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*

Volume 24 **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*

Volume 25 **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*

Volume 26 **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*

Volume 27 **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*

Volume 28 **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection A Framework to Support Experimentally-based Software Engineering Technology Selection*

Volume 29 **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*

Volume 30 **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*

Volume 31 **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*

Volume 32 **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

Volume 33 **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*

Volume 34 **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*

Volume 35 **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*

Volume 36 **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*

Volume 37 **Ansgar Lamersdorf** (2011), *Model-based Decision Support of Task Allocation in Global Software Development*

Volume 38 **Ralf Carbon** (2011), *Architecture-Centric Software Producibility Analysis*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.
In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

ISBN 978-3-8396-0372-7

Fraunhofer IESE

Technische Universität Kaiserslautern
AG Software Engineering

PhD Theses in Experimental Software Engineering