# Chapter IV
# Software Engineering – Processes and Tools

Gerhard Weiss, Gustav Pomberger
Wolfgang Beer, Georg Buchgeher, Bernhard Dorninger, Josef Pichler,
Herbert Prähofer, Rudolf Ramler, Fritz Stallinger, Rainer Weinreich

## Introduction 1

Software engineering traditionally plays an important role among the different research directions located in the Software Park Hagenberg, as it provides the fundamental concepts, methods and tools for producing reliable and high quality software. Software engineering as a quite young profession and engineering discipline is not limited to focus on how to create simple software programms, but in fact introduces a complex and most of the time quite costly lifecycle of software and derived products. Some efforts have been made to define software engineering as a profession and to outline the boundaries of this emerging field of research [PP04, Som04]. Several different definitions of the term software engineering appeared since its first mentioning on a NATO Software Engineering Conference[1] in 1968. A good example of an early definition of the term software engineering which is often cited in the literature is the following:

> The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them. [Boe76]

Another generally accepted definition of software engineeering was given by the IEEE Computer Society:

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). [IEE90]

[1] Proceedings of the famous 1968 and 1969 NATO Software Engineering Workshops are available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html

In a first, joint effort of a scientific and industrial initiative a commonly accepted knowledge base had been created with the goal to define the boundaries of the modern software engineering profession. This "Software Engineering Body of Knowledge" (SWEBOK) [ABD+04] introduces the following ten relevant Knowledge-Areas (KAs):

- Software requirements
- Software design
- Software construction
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering tools and methods
- Software quality

In fact, empirical studies show that software products continue to reflect significantly lower quality than other industrial products and that software projects fail significantly more often than other projects. This clearly indicates that software production has not yet advanced from a handicraft to industrial production. As state-of-the art research in the software domain shows, this raises a very broad spectrum of research issues and themes. Software production can not longer be seen as an activity where a single gifted developer implements an unmated algorithm. Instead, to develop software products means to follow a clear process composed of a large number of possibly complex activities which may be spatially and temporally distributed across multiple development teams. Software continues to grow in size and complexity, and software is involved in most everyday activities, such as making a phone call, making payments and driving a car. An IEEE article titled "Why software fails" [Cha05] states that a typical mobile phone contains around two million lines of software code, and according to General Motors the software used in a car will have 100 million lines of code in 2010. This article also highlights the fact that failing software and IT projects tend to seriously hazard the economic activity of companies, as software typically implements and automates essential business activities. Identified reasons for this failure are as follows:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity

- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

As several of these reasons indicate, a structured and precise defined set of requirements represents the basis for communication about the development progress, planned quality or changes within the products features over time. One of the major responsibilities of a software project manager is to balance the features and quality of software and the time and costs of development. Software development projects often fail according to an unbalanced focus on a specific aspect at the expense of others (e.g., a focus on the number of demanded software features at the expense of the quality of the software to be developed). Significant progress has been made over the past decades in establishing methods and techniques that help to master demands on software and the software development process. However, as the theory and practice of software engineering shows, a pressing need remains for improving these methods and techniques, inventing new ones, and proving their relevance in an industrial context.

This chapter overviews selected R&D activities conducted in response to this need at Software Competence Center Hagenberg GmbH (SCCH, http://www.scch.at) during the previous years. The chapter's focus is on four related key topics in software science which are of particular practical relevance:

- the software development process (Section 2)
- the quality of software (Section 3)
- architectures for software systems (Section 4)
- domain-specific languages and modeling (Section 5)

The work described in this chapter has been done in close cooperation with the Institute of Business Informatics – Software Engineering (Prof. Pomberger) and the Institute of System Software (Prof. Mössenböck) from Johannes Kepler University (JKU) Linz, which both are long-standing scientific partners of SCCH, with the intention to bridge the gap between latest scientific know-how in the area of software engineering on the one hand and software challenges companies are confronted with in their everyday business on the other hand.

## Software Process Engineering 2

Informally spoken, a *software process* is the set of activities, methods, and practices that are used in the production and evolution of software. IEEE de-

fines a *process* as "*a sequence of steps performed for a given purpose*" [IEE90] and more detailed a *software development process* as "*the process by which user needs are translated into a software product (. . . )*". For our purposes a *software process* can be defined as a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated work products (see [PCCW93]). It is important to note that we use the term *software process* intentionally in a broad view referring not only to an organization's overall software process, but to any process or sub-process used by a software project or organization and any identifiable activity that is undertaken to produce or support a software product or software service. Besides planning, designing, coding, testing this view thus also includes activities like estimating, inspecting, reviewing, measuring, and controlling.

From a business point of view, the processes that software organizations apply to develop products and services play a critical role in the implementation of strategies and related plans and objectives. Organizations that are able to control their processes are able to better predict characteristics of their products and services as well as costs and schedules and can improve the effectiveness, efficiency, and—as a consequence—profitability of their business [FPC97].

From an engineering and scientific point of view, *software engineering* is emerging and maturing as an engineering discipline [Sha90, Was96]. Although it is recognized that due to the great variations among application types and organizational cultures it is impossible to be prescriptive of the software process, the concept of *software process* is seen as one of the pillars of a foundation for a software engineering discipline (see [Was96]). *Software engineering* as an engineering discipline also comprises the scientific treatment of the software engineering process (see e.g. [BD04]) in order to understand and systematically improve the software engineering process so that software systems can be built and evolved with high quality, on time and within budget.

*Software process management*, on the other side, deals with the activities that are essential for managing the processes associated with developing, maintaining, and supporting software products in a way, that the produced products and services adhere to internal and external customer requirements and that they support the business objectives of the organization producing them. Key activities identified to be central to *software process management* are *process definition*, *process measurement*, *process control*, and *process improvement* [FPC97].

The objectives of *software process management* are to ensure that defined processes are followed and performing as expected, and to make improvements to the processes in order to help meeting business objectives. From an individual's perspective, the objective of software process management is to ensure that the processes he/she operates or supervises are predictable, meet customer needs, and are continually being improved. From the overall

organizational perspective, the objective is to ensure that the above objective is fulfilled for every process within the organization (see [FPC97]).

With software becoming more and more important for our daily life at increasing speed and with faster turnover cycles, in particular the field of *software process improvement* as a means to deliver better quality products and increase efficiency and effectiveness of software development has become an important part of the software engineering discipline. As a consequence, the work with software processes has emerged as a field of its own, similarly to product development, but with the proper *software process* as the product that has to be developed, kept up and maintained. As a consequence, it is no longer appropriate to talk about *software process improvement* or *software process management*, but about *software process engineering*, indicating that the same quality models and process improvement and maintenance efforts applied to software product development should be applied to the proper software process work and its work products (see [Kin01]). Consequently the terms *software process management* and *software process engineering* are often used widely synonymously.

SCCH has carried out a series of application-oriented research projects related to the concepts, models and methods of *software process engineering*, namely:

- Project *HighLight* (ongoing): identification of the specific needs of small and very small enterprises with respect to software process improvement; development of a lightweight process improvement methodology; application of a product line approach to software process modeling.
- Project *Hephaistos* (ongoing): identification of best practices for the integration of business-driven product lifecycle management and engineering-focused product development in multi-product and product family contexts; current focus on the integration of product management, requirements management and architecture mangement, and change impact analysis.
- Project *GDES-Reuse*: enhancement of a process reference and process assessment meta-model to integrate the concepts of continuous process capability and staged organizational reuse maturity; identification of the relationship of reuse paradigms; development of a process reference model for reuse and of an organizational reuse maturity model for industrial engineering; development of methods for reuse assessment and reuse improvement measure identification and planning.
- Project *SISB*: development of a methodology for the evaluation of engineering strategies as reference framework for process management; identification of "strategy objects" for industrial engineering; mapping and validation of "strategy objects" against best practice process areas.

The remainder of this section shortly provides an overview on relevant concepts related to the field of software process engineering. Based on this, we present details and results of the above listed projects by identifying se-

lected research challenges within the field of software process engineering and demonstrating how the results of the projects contribute to tackling these challenges.

## 2.1 Concepts Related to Software Process Engineering

This subsection shortly introduces selected key concepts related to software process engineering in order to facilitate the understanding of the subsequent subsection on software process engineering research challenges. These key concepts are: levels of models in software process engineering, benefits and importance of software process engineering, relationship "process quality" – "product quality", best practice software process models and process model classification, software process capability and maturity frameworks, and methods for software process evaluation and improvement.

### Levels of Models in Software Process Engineering

Figure 1 depicts the four main levels of models involved in *software process engineering* and identifies the corresponding software process engineering activities associated with model instantiation.

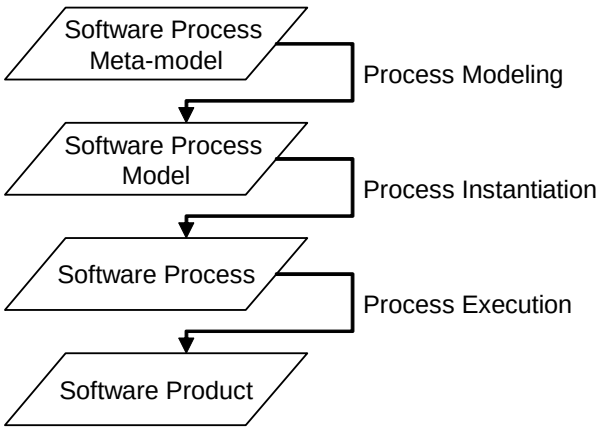



FIGURE 1 Levels of models in software process engineering.

*Software process meta-models* define the "*conceptual framework for expressing and composing software process models*"[Lon93]. They describe the relevant software process sub-models, their basic concepts, and their relationships and define the notation and language for expressing software process models. *Software process models* are the key result of the *process modeling* activity and serve as abstract representations of software processes. They prescribe a software process in terms of the activities to be carried out, the roles and work product types involved, etc. *Software processes* on the other side are the result of a *process instantiation* for a specific software development endeavor. This step often also includes tailoring or customizing of the software process model to the specific needs and goals of a project. The assignment of tools or other instruments for supporting the carrying out of activities and the assignment of resources to roles is typical for this level. The *execution* of a specific software process within a specific software development endeavor finally leads to the creation of the proper *software product* that in this view itself is seen as an instantiation of the software process.

Generally, software processes are complex entities comprised of a number of steps, resources, artifacts, constraints, etc. Depending on the intended usage, software process models reflect this complexity to a certain degree. Key motivators for meta-model-based process modeling are process model reuse, as sharing and composing process models and process model components require proper interfaces, and the automation of process execution. Examples of process meta-models range from simple models that only explain the basic relationships among activity types and result types (e.g. cascade model [Chr92]) to fully-fledged meta-models for software and systems engineering claiming to be capable of instantiating any software process model and method ([Obj08]). [HSSL02] presents a process meta-model and process model concept for component-based software engineering that incorporate the concept of process capability assessment (see "Software Process Capability and Maturity Frameworks" on page 167) into software process modeling. Pointers to examples for software process models can be found in "Best Practice Software Process Models and Process Model Classification" on page 165.

### Benefits and Importance of Software Process Engineering

Software process models play an important role in software engineering. They allow the separation of process aspects from product aspects and provide a basis for the unification of methods and tools. Further general goals pursued with the use of software process models encompass the facilitation of the understanding and communication of the process, the establishment of a basis for the automation of software engineering activities and the establishment of a basis for analyzing and improving an organization's software process (see e.g. [GJ96]). Moreover, the use of software process models provides a series of

further, less quantifiable benefits, like better teachability of the process and easier familiarization of new employees with an organization's practices and procedures, increased independence of specific persons and the establishment of a general basis for professionalism and credibility.

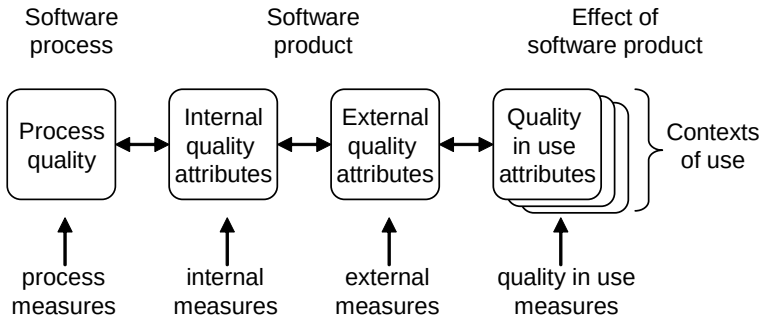
Motivated by the overall goal of enhancing performance, improving an organization's software process has become a central topic in software process engineering. Research shows that improving an organization's process quality can lead to substantial gains in productivity, early defect detection, time to market, and quality, that in total add up to significant returns on the investment in software process improvement. Further identifiable benefits refer to improved cost performance, improved estimates and deliveries on schedule, and increased customer as well as employee satisfaction [HCR+94].

### Relationship "Process Quality – Product Quality"

*Software quality* is generally regarded a key to economic success for software developing organizations and has been an issue since the early days of software development. Consequently, a serious of definitions of software quality from many different viewpoints has emerged. According Garvin five such major approaches to the definition of quality can be identified, namely: the transcendent approach of philosophy, the product-based approach of economics, the user-based approach of economics, marketing, and operations management, and the manufacturing-based and the value-based approaches of operations management [Gar84].

Incorporating these different viewpoints for software engineering, ISO/IEC 9126 [ISO01] and ISO/IEC 25000 [ISO05] provide a *quality model for software* that at the top-level decomposes software quality into the aspects of *process quality*, *product quality*, and *quality in use* (see Figure 2). Product quality, in this context, is determined by the degree to which the developed software meets the defined requirements, while quality in use addresses the degree to which a product is fit for purpose when exposed to a particular context of use (see Section 3 on "Software Quality Engineering" for more details).

*Process quality* within this quality model expresses the degree to which defined processes were followed and completed and assumes that software processes implement best practices of software engineering within an organization. The basic assumption that the quality of a software product is largely influenced by the process used to develop it and that therefore, to improve the quality of a software product, the quality of the software process needs to be improved is the underlying principle of a series of software process capability and maturity frameworks and related methods for software process evaluation and improvement (see subsections below). It is meanwhile explicitly recognized in the international standard on "Software product Quality Requirements and Evaluation" (SQuaRE) together with the assumption that the




Software quality model according ISO/IEC 25000. FIGURE 2

*quality of a software process* is the extend to which this process is explicitly defined, managed, measured and continuously improved [ISO05]. [SDR+02] describes an approach to component based-software engineering, which explicitely adds process capability information to the quality information of software components.

## Best Practice Software Process Models and Process Model Classification

The history and evolution of *software process models* dates back to the very beginning of software engineering. While early models suggested a *code and fix cycle*, the first major milestone in software process modeling is the *waterfall development model* [Roy70], that groups activities into major phases based on the ideal assumption of sequential execution, but does not explicitly foresee changes or a prototyping oriented development approach. To overcome these weaknesses the *spiral model* [Boe88] suggests a cyclical, risk-driven development approach, in which - before entering a new cycle - the project risks are analyzed and appropriate measures taken. More recent examples of industry driven software process models are iterative and incremental models like the *Rational Unified Process (RUP)* [Kru03] or its open source subset *Open Unified Process (OpenUP)* [Ope08] that also incorporate principles of agile development.

Further, *standardization* plays a major role in software process modeling. An example of a quasi-standard is the *German V-Model*, which considers the development of a software system from the side of the supplier as well as from the side of the acquirer. It is organized with a focus on work products and provides work product quality standards and state models and also foresees

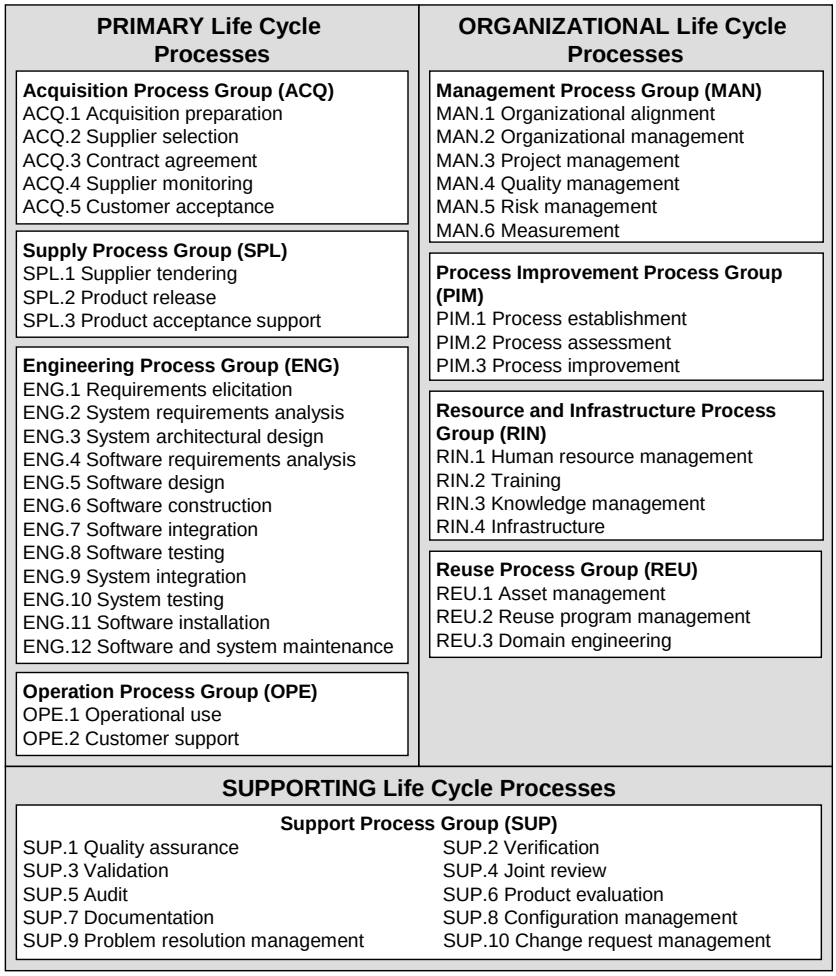



FIGURE 3 Software life cycle processes according ISO/IEC 12207.

predefined project execution strategies depending on predefined project types [V-M06]. At an international level the standard ISO/IEC 12207 [ISO95] on *software life cycle processes* provides a best practice software process model (see Figure 3) that also includes organizational and supporting processes. In the form presented in AMD1 and AMD2 to the standard this model also serves as a process reference model for process capability evaluation.

The available software process models vary significantly regarding the level of detail provided and the project or organizational scope covered by the processes defined. Regarding the *level of detail*, [Hum89] distinguishes between

*universal level models* providing general guidelines, principles and policies as high-level framework, *worldly level models* providing procedures that implement policies at working level and practices that guide daily work, and *atomic level models* providing detailed refinements like standards, tools, techniques, etc. that are appropriate for process automation.

A further important distinction of software process models at conceptual level in the context of software process evaluation and improvement is the classification into *process implementation models*, *process assessment models*, and *process reference models*. While *process implementation models* provide the necessary details and guidance to be directly instantiated for process execution (see Figure 1), *process reference models* serve for the evaluation of process capability and benchmarking purposes of the actually implemented processes and—informally spoken—define the requirements for process implementation models from a best practice perspective. *Process assessment models*, finally, break down the requirements of process reference models into measureable indicators of practices and work products and are used within a process evaluation.

## Software Process Capability and Maturity Frameworks

A promissing means to continuously improve the software development process is to regularly evaluate the software process against some kind of best practice based measurement scale. Also a number of so-called assessment-based methods for software process improvement and corresponding measurement scales have been developed, there are essentially two types of such measurement concepts:

*Staged models of process maturity:* These models define a maturity scale at *organizational level* and typically relate each of the levels within the maturity scale to recommended practices necessary for achieving this level.

*Continuous models of process capability:* These models measure the software process at *process level* along two axes—the *process dimension* comprised of a number of processes subject to measurement and typically defined in a *process reference model*, and the generic process *capability dimension* (see Table 1) comprised of process attributes and process capability levels that are applied to characterize each process of the process dimension.

The *Capability Maturity Model (CMM)* [Hum95, PCCW93] is a typical model of the staged type. The approach of continuous process capability was developed within the *SPICE project* [Dor93] and has been standardized within the international standard series ISO/IEC 15504 on *Information Technology – Process Assessment* [ISO03]. According their underlying meta-models, staged models typically provide a predefined road map for organi-

| | |
|---|---|
| **Level 1** | **Performed Process** |
| PA 1.1 | Process Performance Attribute |
| **Level 2** | **Managed Process** |
| PA 2.1 | Performance Management Attribute |
| PA 2.2 | Work Product Management Attribute |
| **Level 3** | **Established Process** |
| PA 3.1 | Process Definition Attribute |
| PA 3.2 | Process Deployment Attribute |
| **Level 4** | **Predictable Process** |
| PA 4.1 | Process Measurement Attribute |
| PA 4.2 | Process Control Attribute |
| **Level 5** | **Optimizing Process** |
| PA 5.1 | Process Innovation Attribute |
| PA 5.2 | Process Optimization Attribute |

TABLE 1 Process capability levels and attributes according ISO/IEC 15504.

zational improvement by identifying improvement priorities generally true for most software organizations. Continuous models, on the other hand, do not prescribe any particular improvement path except the evolution of single processes, but come up with a customizable process dimension.

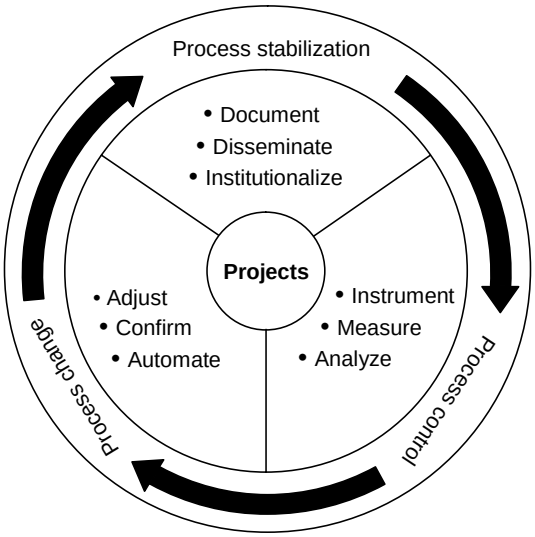
## Models for Software Process Evaluation and Improvement

Beside the process engineering activities built into software best practice process models as own processes or activities, two major types of models for software process evaluation and improvement can be distinguished regarding the issue of scale of the intended improvement activity [Kin01]:

- Software process improvement action life cycle models
- Software process improvement program life cycle models.

*Software process improvement action life cycle models* are primarily meant for guiding a single improvement action and generally fail to give the necessary guidelines for a full software process improvement program. As they do not address improvement program-level issues, they are typically kept relatively simple. Examples of such models are:

- the Plan-Do-Check-Act (PDCA) model [She31]
- the Process Improvement Paradigm-cycle [Dio93] (see Figure 4).




Process improvement paradigm cycle. FIGURE 4

Models of this type are primarily intended for software process staff, process owners, and non-process professionals having a role in a software process improvement action.

*Software process improvement program life cycle models* on the other side, put more emphasis on aspects such as initiation, management and coordination of the overall improvement program and in particular on the coordination of individual process improvement actions. Examples of such models are:

- the IDEAL (Initiating-Diagnosing-Establishing-Acting-Learning) cycle [McF96] and
- the ISO 15504-7 cycle [ISO98] (see Figure 5).

These models are mainly intended for people who have been entrusted the management of a large scale process initiative. They are important for staging and managing a successful improvement program and represent a major step towards an institutionalized software process engineering system.

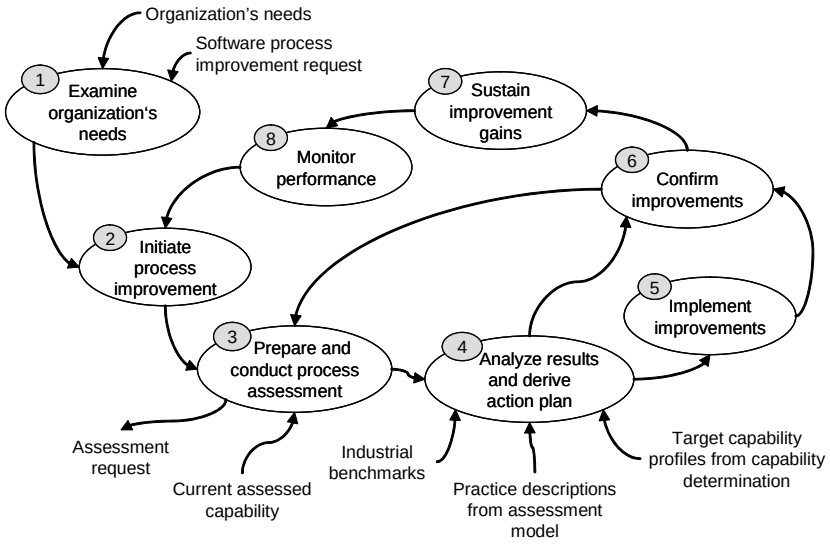



FIGURE 5 ISO/IEC 15504-7 process improvement cycle.

## 2.2 Software Process Engineering Research Challenges and Application-oriented Research at SCCH

In this section we present selected challenges within the field of *software process engineering* that are tackled by research projects performed at SCCH together with partner companies. The challenges dealt with are:

- Software process improvement for small and very small enterprises
- Integration of product engineering and lifecycle management
- Integrating process capability and organizational reuse maturity
- Alignment of process improvement with strategic goals

### Software Process Improvement for Small and Very Small Enterprises

The project *HighLight* is an ongoing research project focusing on the specific needs and requirements of small and medium sized enterprises (SMEs) with respect to software process improvement. Specific attention is devoted to very small enterprises as a relevant subset of SMEs.

SMEs constitute a significant part of the Austrian as well as European industry. In particular software-oriented SMEs, i.e. either SMEs that develop software for customers or SMEs for which software developed in-house represents an essential part of their products or services are confronted with the need to improve the quality of their products in general and of software in particular and to react quickly to changing market and customer needs.

From a software engineering perspective such small development companies have to cope with challenges quite similar to that of large companies, like the need to manage and improve their software processes, to deal with rapid technology advances, to maintain their products, to operate in a global environment, and to sustain their organizations through growth. In the case of small companies however, different approaches to tackle these challenges are required because of specific business models and goals, market niche, size, availability of financial and human resources, process and management capability, and organizational differences [RvW07].

Over the last twenty years the software engineering community has paid special interest to the emerging field of software process improvement as a means to increase software product quality as well as software development productivity. However, the rise of software process improvement was primarily driven through its successful implementation in large companies and there is a widespread opinion stressing the point that the success of software process improvement is only possible for large companies. Nevertheless, within the last five years the software engineering community has shown an increasing interest in tackling the software process improvement challenge for small companies and there is a growing interest of the software engineering community to adapt the models and methods developed for software process improvement in the large to the specific needs of SMEs. [PGP08] presents a systematic review on software process improvement in small and medium software enterprises through analysis of published case studies. The challenge to provide systematic support for software process improvement to small and in particular very small enterprises is meanwhile also subject to international standardization efforts (see [ISO09]).

The overall goal of *HighLight* is therefore to research into, improve and develop innovative concepts, models and methods for the identification and efficient transfer of software engineering best practices to small and medium sized software enterprises. The specific goals and pursued results include:

- the identification of the specific needs, constraints and expectations of SMEs with respect to software process and quality improvement;
- the investigation into the state of the art in software process and quality improvement approaches, methods and standards, particularly with respect to their applicability to SMEs;
- the investigation into the reasons for success or failure of software process and quality improvement initiatives at SMEs and the identification of the critical success factors for such endeavors;

- the compilation of a comprehensive pool of software engineering best practices or pointers to those covering the specific contexts of SMEs;
- the development of a lightweight software process and product improvement methodology specifically targeted for SMEs and particularly tailorable to their strategic and business needs and project and product contexts;
- the validation of the developed concepts, models and methods in selected real-world improvement projects;
- the setup of a forum for discussion and exchange for SMEs interested in software process and quality improvement.

The work within *HighLight* is explicitly based on and driven by the needs and constraints of SMEs with respect to software process and quality improvement. The project builds on existing experience and integrates a process, product and business view on software process and quality management into a lightweight improvement methodology for SMEs. The project focuses on identifying selected development phase, paradigm and—where appropriate—technology-specific software engineering best practices and compiling them into compact, modular and integrated process reference models particularly suited for SMEs. The work will also include the development of concepts and methods for supporting the evolution of an organization's processes through changing paradigms and evolving organizational contexts, in particular company growth and increasing software productization.

As a means to implement such an approach the study of the applicability of concepts from the software product line area [CN02] to software process engineering and the creation of a *"software process line"* for SMEs is envisioned. In such an approach—in analogy to software product features used in software product line approaches to determine the concrete software product within a software product line—the characteristics that describe the organizational, project, product and market context of a software organization are envisioned to determine the software process model out of a *software process model line* that is most appropriate for the respective organization. An initial literature research shows that a similar approach—in a limited scope—has so far only been applied for the definition of project-specific processes for hardware/software co-design in an embedded system domain (see [Was06]).

Furthermore, *HighLight* concentrates on identifying lightweight and efficient methods for transferring the identified best practices into SMEs and will also seek feedback on and try to empirically validate the developed concepts and methodologies. *HighLight* will therefore seek liaison with software development projects at SCCH and within SCCH's partner companies and will establish links with established communities and institutions in the field of software process and quality improvement. Particularly SMEs that generally do not possess any kind of organization development departments will gain a lightweight, efficient and effective method that supports them in adapting to high-quality, state-of-the-art, lean and efficient software engineering processes.

### Integration of Product Engineering and Lifecycle Management

The development of software products today is strongly driven by business considerations and market forces throughout the whole lifecycle. Traditional project-focused software development emphasizes distinct project phases and a functionally separated organization with distinct and specialized roles. As a consequence, problems emerge from different stakeholder perspectives on the same underlying product, locally optimized processes, isolated tools, and redundant, inconsistent and often locally stored product data. In real-world contexts these problems are additionally increased, as software organizations often have to manage a number of interrelated and interplaying products and services or as software is only part of the offered product or service. Key challenges in the management of software product development are thus to align the different perspectives to the overall business objectives; to establish consistent and integrated processes, methods and tools that span the different groups; to manage the relationship between the development artifacts produced by the different groups and processes; and to monitor product development progress across the whole lifecycle. The ongoing project *Hephaistos* carried out together with two local project partners tackles these challenges by identifying best practices for the integration of business-driven product lifecycle management and engineering-focused product development.

In this subsection we report about initial, interim results of *Hephaistos* with respect to software process engineering and provide an outlook on further research to be carried out in this context. These interim results are:

- an identification of key problems related to the integration of product engineering and lifecycle management,
- the identification of solution concepts addressing the identified problems, and
- the development of a conceptual model relating the identified solution concepts to core software engineering and management activities.

By analyzing the organizational context of one of the partner companies we identified a number of company-specific problems that were generalized into key problem areas, that also correspond to and are confirmed through issues and obstacles observed in many other industrial projects. These key problem areas related to the integration of product engineering and lifecycle management are [PRZ09]:

- lack of reuse of lifecycle artifacts beyond code, e.g. of requirements, design,
- unclear rationale regarding past informal and undocumented decisions, e.g. regarding requirements or architecture,
- intransparent consequences of changes, in particular during software product maintenance and enhancement,
- imbalance between management oriented lifecycle activities and overemphasised core engineering activities,

- heterogeneous tool-infrastructures lacking integration and interoperability and as a consequence hampering collaboration of roles,
- disruption of workflows and processes as a consequence of limited interoperability of tools that lead to redundant activities and data and increased overhead, error-prone work, and inconsistencies,
- intransparent status of artifacts and work progress due to heterogeneous tool landscapes,
- inability to reconstruct past states, in particular of non-code lifecycle artifacts, and
- missing integration of product management and project management, in particular when software products evolve over years and are maintained and enhanced in a series of ongoing, parallel projects.

Application lifecycle management (ALM) promises to tackle a wide spectrum of the above challenges and over the last years a large number of ALM solutions have been announced. However, the term is quite new, lacks a common understanding and is mostly driven by tool vendors in order to emphasize their move towards integrated tool suites covering the whole application lifecycle.

In order to help analyze and compare these solutions objectively, we identified two main goals of ALM by matching the tool vendors' propositions with the key problem areas listed above. These goals are:

1. Seamless integration of engineering activities at tool and process level across the whole lifecycle of an application;
2. Emphasis on management activities to shift the technical perspective of engineering towards the business perspective of software management.

We then further refined these two main goals into the following solution concepts for the integration of product engineering and lifecycle management [PRZ09]:

*Traceability:* Traceability is defined as "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another" [IEE90].

*Version control:* Over the lifecycle of an application multiple versions evolve and require consistent control for managing releases, maintaining defined states and baselines across different artifacts, as well as allowing reverting to these defined states.

*Measurement:* Retrieving information about products, processes and resources as well as their relationships is the basis for establishing transparency, objective evaluation and planning. The role of measurement is essential for the management of software projects [DeM86] and has to be expanded to the entire application lifecycle.

*Workflow support:* Workflows bring together a sequence of operations, resources, roles and information flows to achieve a result. Approaches are

necessary that provide interoperability of tools and processes to establish workflows across the entire lifecycle of an application.

*Collaboration support:* As software development is a team endeavor, concepts and tools for collaboration have found their way into software development.

*Shared services:* In addition to the above solution concepts a number of further basic services were identified like managing users and access rights, etc. which are relevant for every activity and tool applied.

The solution concepts identified above have a strong focus on integration in two directions: firstly, they provide support for the integration of engineering activities over the whole application lifecycle at tool level as well as at process level, addressing the first goal; secondly, the same concepts also enable the integration of engineering activities with management activities by establishing the link between the technical and the business perspective, addressing the second goal. Figure 6 depicts these three dimensions showing the engineering activities and the management activities according to [BD04] on the x-axis and the y-axis respectively. The solution concepts identified above are depicted as third dimension to clearly distinguish them from engineering and management activities and to highlight their role of tying the different engineering and management activities together.

The model depicted in Figure 6 by now has been used to support the definition of a strategy for improving the integration of product engineering and lifecycle management activities and for process improvement in the analyzed company. The strategy is subject to implementation, with a current focus on the integration of product management, requirements management and architecture design. Preliminary results confirm the applicability of the model as guidance for identifying and prioritizing problem areas as well as planning for a tailored ALM solution.

## Integrating Process Capability and Organizational Reuse Maturity

The goal of the project *GDES*$^2$*-Reuse* that we carried out together with *Siemens Corporate Technology* was the development of an assessment-based methodology for evaluating an industrial engineering organization's reuse practices and identifying and exploiting its reuse potential.

While software engineering deals with software only, industrial engineering has to enable the parallel development of different engineering disciplines, like mechanical engineering, electrical engineering, and communications and control system engineering. Industrial engineering projects range from rather

[2] Globally Distributed Engineering and Services

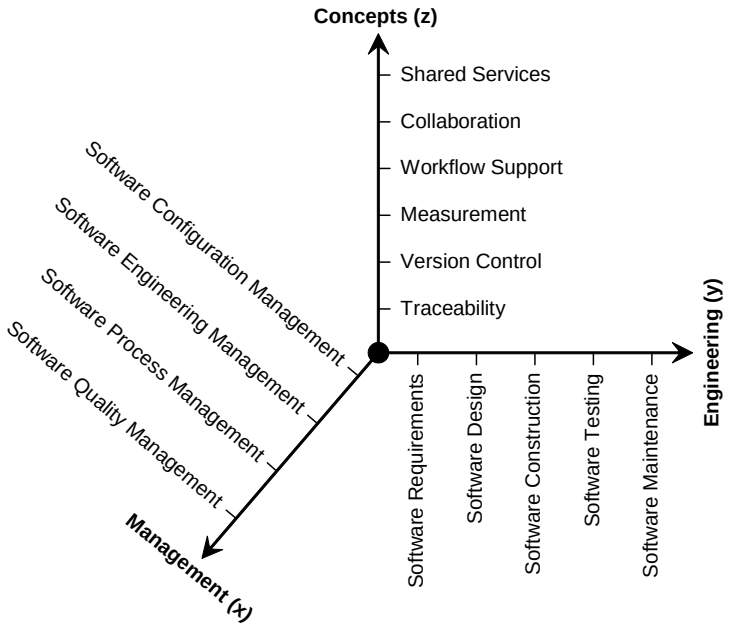



FIGURE 6 Conceptual model for product engineering and lifecycle management integration.

simple and small projects (e.g. semi-automated assembly line) to large and highly complex projects (e.g. nuclear power plants). Like software engineering, industrial engineering today has to cope with increasing demands for more flexible, more reliable, more productive, faster, and cost optimized planning and realization of industrial solutions. Simultaneously, industrial engineering has to deal with more demanding customer requirements, increased complexity of solutions and harder competition in a global market. Increasing reuse has therefore been identified as one key element for increasing quality and productivity in industrial engineering (see [LBB+05]). Reuse is one of the most basic techniques in industrial engineering and pervades all engineering phases and all engineering artifacts. Although recognized as a fundamental and indispensable approach, it is hardly systematized and often only applied in an ad hoc manner. As a consequence the reuse potential in industrial engineering organizations is rarely exploited and in most cases not even known.

On the other side, reuse is well understood in the domain of software engineering (see e.g. [JGJ97, Sam01, MMYA01]) and the distinction between bottom-up reuse concepts like component-oriented reuse and top-down approaches like copy-and-modify, reuse of prefabricates (e.g. application frameworks), the application of platforms, or the system-family or software product

line approach [CN02] is well established. In the context of industrial engineering top-down approaches are particularly interesting, as they imply that the reusing organization has a general understanding of the overall structure of an engineering solution.

Our core development work for the overall reuse improvement methodology was preceded by the evaluation of relevant process and product evaluation approaches and standards and respective models and meta-models as well as reuse improvement approaches. This resulted in the selection of a process-centered approach, focusing on the identification of best practices for reuse in industrial engineering based on an adaption of the meta-model of ISO/IEC (TR) 15504 for process reference models and process assessment models as implicitly defined in [ISO03].

The methodology for improvement of reuse in industrial engineering developed within *GDES-Reuse* is intended to be applicable to all kinds of organizations and market segments of industrial engineering and is comprised of three sub-methodologies that are partly also independently applicable:

- a methodology for the *evaluation of the actual situation* of an engineering organization with respect to reuse, that allows to assess to what extent the respective organization fulfills the identified reuse best practices, focused around three distinct, but interplaying and interrelated models: the *process reference model*, the *reuse maturity model*, the *assessment model for reuse* in industrial engineering,
- a methodology for *potentials analysis* that—based on the results of the evaluation and under consideration of external factors and organizational goals—supports the identification of an optimal set of reuse practices for the respective organization, and
- a methodology for *action planning* that—based on the results of the evaluation and the potentials analysis—identifies and prioritizes the necessary measures for introducing or improving reuse.

Table 2 provides an overview of the *process reference model*. The model defines the results necessary for successful reuse and organizes these results according the typical phases of the engineering life cycle which are themselves grouped into categories. The overall objective is to support the representation of evaluation results and to make them comparable across organizational boundaries. Further details on the categories and phases can be found in [SPP+06].

The *reuse maturity model* for industrial engineering defines the results necessary for successful reuse. Based on [PRS00], it organizes these results into distinct organizational reuse maturity stages that build one upon the other in order to provide general guidance for the introduction and improvement of reuse within an engineering organization. The model foresees four maturity stages that are characterized in Table 3.

The *assessment model*, finally, breaks down the reuse results into reuse base practices and input and output artifacts that are used as indicators during evaluation.

| Contracting (CON) | Engineering for Reuse (EFR) |
|---|---|
| CON.1 Acquisition/Initiation<br>CON.2 Customer Requirements Analysis<br>CON.3 Bid Preparation<br>CON.4 Customer Acceptance | EFR.1 Domain Analysis<br>EFR.2 Domain Design<br>EFR.3 Domain Implementation<br>- EFR.3.1 Domain Impl. - Discipline<br>- EFR.3.2 Domain Impl. - Integration |
| **Engineering with Reuse (EWR)** | **Organizational Support of Reuse (OSR)** |
| EWR.1 System Requirements Analysis<br>EWR.2 Basic Engineering<br>EWR.3 Detail Engineering:<br>- EWR.3.1 Detail Eng. - Discipline<br>- EWR.3.2 Detail Eng. - Integration<br>EWR.4 Realization and Operational Test<br>EWR.5 Start of Operation<br>EWR.6 Maintenance and Servicing | OSR.1 Reuse Program Management<br>OSR.2 Improvement of Reuse<br>OSR.3 Measurement of Reuse<br>OSR.4 Asset Management<br>OSR.5 Quality Assurance<br>OSR.6 Change Management<br>OSR.7 Problem Resolution |

TABLE 2 Structure of the reference model for reuse in industrial engineering.

| |
|---|
| **1 – Chaotic:** Reuse is done ad-hoc only and not systematically. If needed, artifacts from previous projects are used as starting point for new ones. Reuse takes place unplanned, uncoordinated, undocumented, informal, occasional, and local and randomly on a small scale. Form and degree heavily depend on persons. Its contribution to achieving business goals is limited. |
| **2 – Systematical:** Reuse is pursued systematically. The technical and organizational measures for structured reuse are in place. Solutions are designed modular and the reuse of artifacts is supported by in-house development, purchasing and documentation of artifact usage. Reuse of artifacts is based on conformance with industry specific standards as well as definition and compliance with internal standards or interfaces. |
| **3 – Domain-oriented:** The domain specific benefits of reuse are exploited. The business is analyzed and reusable artifacts are defined based on the analysis of recurring requirements. Reusable artifacts are thus customized to the business domain. Reuse is supported by organization and processes. An organization wide infrastructure for reuse is in place and planning, coordination and controlling of a reuse oriented engineering process is established. Domain specific reference architectures are typical at this stage. |
| **4 – Strategic:** The whole organization is strategically oriented towards reuse. Reuse is performed systematically and integrated across all phases of the engineering life cycle. This is reflected in the business strategy and in the orientation of all business functions towards reuse, including marketing, sales, acquisition, etc. The portion of reused artifacts is high, as well as the contribution of reuse to achieving business goals. |

TABLE 3 Characteristics of reuse maturity stages.

The core objective of the *potentials analysis* methodology is to derive an optimal reuse-oriented target scenario for the engineering processes within the assessed organizational unit based on the organizational unit's business and organizational goals, evaluation results, and exogenous factors like customer or market requirements, available or future technologies, characteristics of competition, etc. The potentials analysis methodology represents the link between the evaluation of the current situation regarding reuse and the method for action planning for improvement of reuse. It serves to identify highly rewarding and not yet implemented reuse practices for subsequent action planning for the implementation of these practices and is rather strategically and tactically oriented. The identification of improvements for reuse in industrial engineering is much more complex compared to "traditional" capability-oriented process improvement. Orthogonal to improving along the process capability dimension it also involves strategic decisions on the overall design of the engineering process, the pursued engineering and reuse paradigms, the desired organizational reuse maturity stages, etc. More details on the potentials analysis method and the related action planning method can be found in [SPPV09].

The methodology deliverables briefly described above are based on the establishment of a conceptual framework through the enhancement of existing meta-models in order to integrate the concept of organizational reuse maturity with the concept of continuous process capability that itself represents a sigificant achievement of the GDES-Reuse project [SPP+06]. All three models, the process reference model (PRM), the reuse maturity model (RMM), and the process assessment model (PAM) for reuse in industrial engineering, capture reuse best practices at different levels of abstraction and organize and represent them from different points of view (see Figure 7):

The PRM as well as the RMM contain the same set of reuse results in the sense of ISO/IEC 15504 process outcomes. While the PRM organizes these reuse results by phases of the engineering life cycle which are themselves grouped into categories of phases, the RMM organizes these reuse results into stages of organizational reuse maturity. The PAM on the other hand picks up the set of reuse results as defined in the PRM and RMM together with the organization of these reuse results by phases from the PRM and breaks down these reuse results into reuse base practices and input and output artifacts as indicators during evaluation. Reuse results represent the core conceptual element of the *GDES-Reuse* methodology providing the bridge between the continuous PRM and the staged RMM and in consequence between the evaluation methodology and the methodology for potentials analysis. From a meta-model point of view the PRM is fully compliant to the requirements of ISO/IEC 15504 [ISO03] for process reference models. It can be interpreted as a partial model of the overall engineering life cycle containing and describing those processes or parts of processes relevant for successful reuse. Consequently the measurement framework for process capability as defined in ISO/IEC 15504 can be directly applied to the reuse results of the PRM

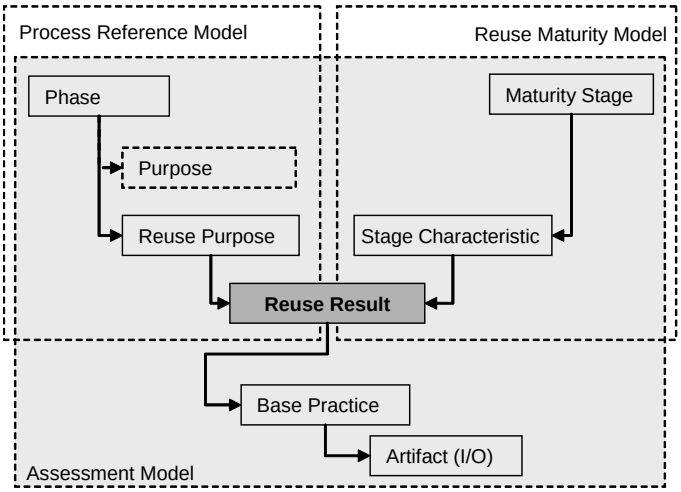



FIGURE 7 Process reference model and reuse maturity model—conceptual framework.

and aggregated towards the phases of the PRM on the one side and towards the maturity stages of the RMM on the other side.

A major value of the work performed within *GDES-Reuse* lies in the integration and systematisation of best practices from a series of reuse approaches in a single model and in the integration of a "staged" reuse maturity model with a "continuous" process model. The focus of the work was on providing a best practice framework for the strategic design of engineering processes in the sense of which paradigm or development approach or combination of those to use. The approach chosen to resolve this problem is compliant to established process assessment and improvement approaches like CMMI [CMM06] or SPICE [ISO03] but much more focused with respect to modelling depth and thus rather a complement to those models than a substitution of those.

Furthermore, we regard the project's results re-transformable and applicable to the domain of software engineering, as the various reuse paradigms and approaches developed in the field of software engineering represented a starting point for model development. Moreover, the engineering of control and communication systems, as one of the core industrial engineering disciplines, typically includes software engineering as a major sub-discipline.

The methodology for the evaluation of an actual reuse situation has so far been applied in two real world evaluation projects (see [SPV07]).

## Alignment of Processes Improvement with Strategic Goals

Under the umbrella of the project *SISB*[3] together with *Siemens Corporate Technology* we carried out research into methods for the *evaluation and development of engineering strategies* for the industrial solutions business. In this section we highlight results from this research that are relevant for the area of process engineering. These main results are:

- an understanding of the role of engineering strategies in the overall strategy development context of an organization,
- the development of a meta-model for describing engineering strategies,
- the identification of the engineering strategy objects relevant for the industrial solutions business, and
- the development of a methodology to support the evaluation and development of engineering strategies.

In order to understand strategy development at the engineering level we have to relate *engineering strategies* to the overall strategy development efforts in an organization. Typically a distinction is made between the *corporate strategy*, various *division strategies* and various *functional strategies* [VRM03]. While a corporate strategy deals with determining which market segments should be addressed with which resources, etc., a division strategy refines the corporate strategy by addressing the major question how to develop a long term unique selling proposition compared to the market competitors and how to develop a unique product or service. Functional strategies on the other side define the principles for the functional areas of a division in accordance with the division strategy and therefore refine the division strategy in the distinct functional areas, like marketing, finance, human resources, engineering, or software development. Depending on the size and structure of a company there might be no explicit distinction between corporate strategies and division strategies, but nevertheless they are part of the relevant context for the development of functional strategies.

Figure 8 depicts the conceptual framework (meta-model) developed for the description of functional strategies. The core elements of such a strategy are *strategic goals*, *strategy objects* and *strategic statements*. The strategic goals formulated in the engineering strategy are refinements of strategic goals on the corporate respectively divisional level, mapped on the functional area. A strategy object is a topic (e.g. process management) that refines one ore more strategic goals. As the strategy objects—and therefore also the strategic statements—are targeted towards the functional strategic goals it is also assured that the divisional or corporate goals are not violated. Although not necessary on the conceptual level, the grouping of strategy objects facilitates understanding of strategy objects on a more abstract level and also allows focusing of the strategy assessment or development process. The approach for

[3] Systematic Improvement of the Solutions Business

grouping strategy objects we finally decided to use, groups strategy objects simultaneously along three dimensions: *strategy key areas* like people, process, products and services, methods and tools; *strategy target groups* denoting the typical responsibility for a strategy object, e.g product management, sales, etc.; and *priority*.

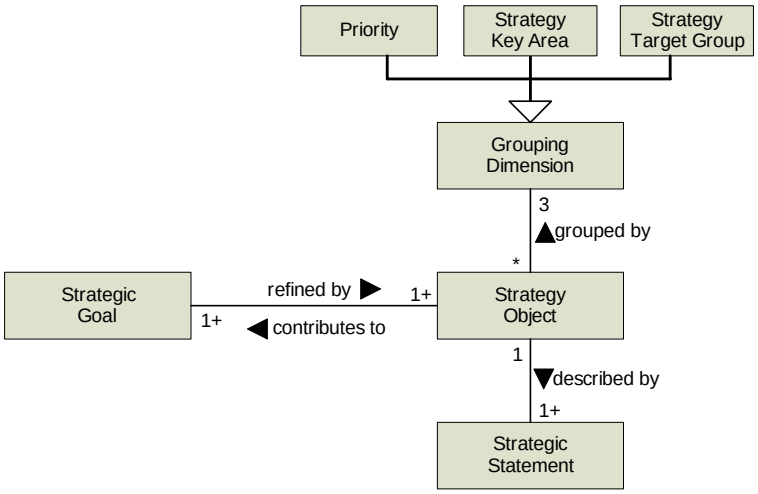
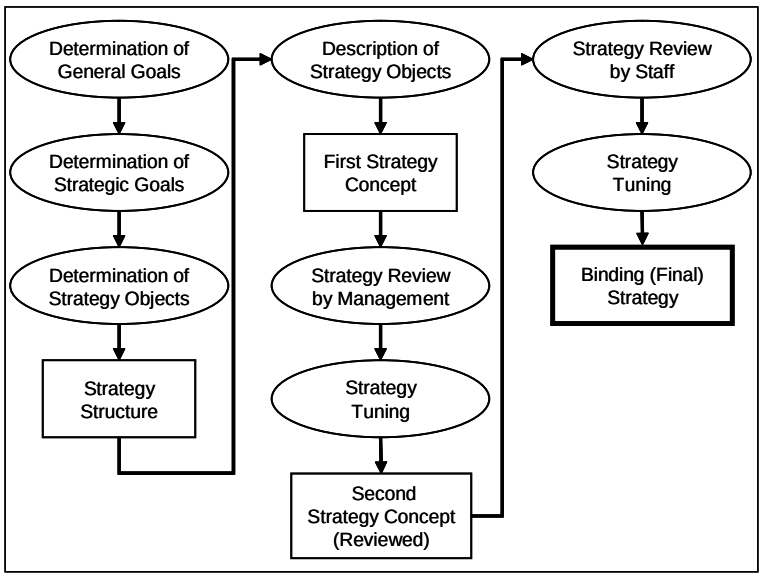



FIGURE 8 Meta-model for describing functional strategies.

*Strategy objects* in the context of an engineering strategy can be understood as a subject area that needs to be dealt with on a strategic level. In order to identify the strategy objects relevant for the industrial solutions business, strategy objects from the software engineering domain were used as a starting point. These were evaluated and adapted for their use in the industrial solutions business and additional strategy objects were identified. These additional strategy objects were identified by analyzing existing functional strategies from engineering organizations. Examples of strategy objects include architecture management, change management, competence management, domain engineering, tool and data integration, process management, quality management, requirements management, reuse management, and standards management. The full list of identified strategy objects together with their definition, identification of typical topics dealt with, examples of strategic statements and the assignment to the three grouping dimensions is provided in [PSN08]. Additionally, it has to be noted that a major step during the strategy development process is to select—and where necessary add—the appropriate strategy objects according to their importance and urgency.

The general approach of the *methodology* for the systematic assessment of existing engineering strategies is to conduct a strategy development process with an assessment emphasis. The typical strategy development process for functional engineering strategies is shown in Figure 9, structured into the development and prioritization of strategic goals, strategy objects and strategic statements.




Process of engineering strategy development. FIGURE 9

As the assessment method simulates parts of a strategy development process the participation of the management responsible for strategy development in a division or company is inevitable. The method itself consists of four main activities:

*Engineering strategy assessment – kickoff:* Determination of the strategy objects relevant for the company or division and identification of information sources.

*Evaluation of strategy objects:* Assignment of existing strategic statements to the selected strategy objects and assessment of the maturity of each strategy object and identification of gaps in the engineering strategy.

*Consolidation of the evaluation:* Adjustment of the assignments of strategic statements as well as of the assessment of the strategy objects together with the responsible management.

*Finalization and presentation of results:* Finalization of the assessment report and management presentation of the results.

The result of this assessment method is a qualitative report indicating the general maturity of the engineering strategy regarding form and structuredness, strategy objects that should have been considered in the engineering strategy, the completeness and maturity of strategic statements for each important strategy object, those strategy objects where existing strategic statements are to weak or few with respect to the relevance of the strategy object, gaps in the engineering strategy in the sense of strategy objects important for the company or division without coverage by strategic statements.

In order to validate the identified strategy objects regarding completeness and coverage of relevant organizational processes, the strategy objects have been mapped against the key process areas of CMMI [CMM06]. As CMMI is a widespread process improvement maturity model for the development of products and services that aims at a wide coverage of engineering disciplines, it was assumed that the process areas described there cover a wide range of organizational processes. The detailed mapping of strategy objects against process aeras is described in [PSN08].

As in particular the *strategy key area "Process"* groups all strategy objects that deal with the management of processes in general, with value chain management, quality management, etc., this grouping allows a customized view on strategic objects and strategic statements from the point of view of process engineering. It thus facilitates capturing and understanding the strategic constraints for the process engineering activity as set by the engineering strategy of an organization.

# 3 Software Quality Engineering

At about the same rate as software systems have been introduced in our everyday life, the number of bad news about problems caused by software failures increased. For example, last year at the opening of Heathrow's Terminal 5, in March 2008, technical problems with the baggage system caused 23.000 pieces of luggage to be misplaced. Thousands of passengers were left waiting for their bags. A fifth of the flights had to be cancelled and—due to theses problems—British Airways lost 16 million pounds. An investigation revealed that a lack of software testing has to be blamed for the Terminal 5 fiasco (ComputerWeekly.com[4], 08 May 2008).

In August 2003 a massive blackout cut off electricity to 50 million people in eight US states and Canada. This was the worst outage in North

[4] http://www.computerweekly.com/Articles/2008/05/08/230602/lack-of-software-testing-to-blame-for-terminal-5-fiasco-ba-executive-tells.htm

American history. USA Today reported: "FirstEnergy, the Ohio energy company ... cited faulty computer software as a key factor in cascading problems that led up to the massive outage." (USA Today[5], 19 Nov 2003).

These and similar reports are only the tip of the iceberg. A study commissioned by the National Institute of Standards and Technology found that software bugs cost the U.S. economy about $59.5 billion per year [Tas02]. The same study indicates that more than a third of these costs (about $22.2 billion) could be eliminated by improving software testing.

The massive economic impact of software quality makes it a foremost concern for any software development endeavor. Software quality is in the focus of any software project, from the developer's perspective as much as from the customer's. At the same time, the development of concepts, methods, and tools for engineering software quality involves new demanding challenges for researchers.

In this chapter we give an overview of research trends and practical implications in software quality engineering illustrated with examples from past and present research results achieved at the SCCH. Since its foundation, SCCH has been active in engineering of high quality software solutions and in developing concepts, methods, and tools for quality engineering. A number of contributions have been made to following areas, which are further elaborated in the subsequent subsections.

- Concepts of quality in software engineering and related disciplines.
- Economic perspectives of software quality.
- Development of tool support for software testing.
- Monitoring and predicting software quality.

## 3.1 Concepts and Perspectives in Engineering of Software Quality

### Definition of Software Quality

Software quality has been an issue since the early days of computer programming [WV02]. Accordingly a large number of definitions of software quality have emerged. Some of them have been standardized [IEE90][6], but most of them are perceived imprecise and overly abstract [Voa08]. To some extent, this perception stems from the different viewpoints of quality inherent in

[5] http://www.usatoday.com/tech/news/2003-11-19-blackout-bug_x.htm

[6] The IEEE Standard 610.12-1990 defines software quality as *"(1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations."*

the diverse definitions. As a consequence, the ISO/IEC Standard 9126:2001 [ISO01] and its successor ISO/IEC Standard 25000:2005 [ISO05] decompose software quality into process quality, product quality, and quality in use. The standard recognizes software as product and reflects Garvin's general observation about different approaches to define product quality [Gar84].

*Process quality:* Software processes implement best practices of software engineering in an organizational context. Process quality expresses the degree to which defined processes were followed and completed.

*Product quality:* Software products are the output of software processes. Product quality is determined by the degree to which the developed software meets the defined requirements.

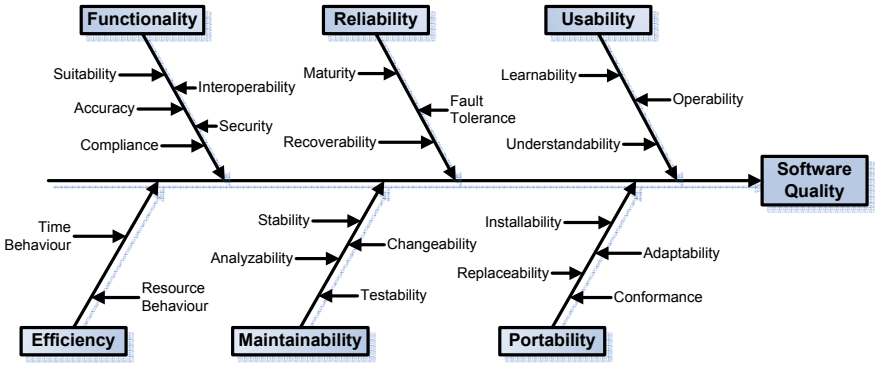
*Quality in use:* A product that perfectly matches defined requirements does not guarantee to be useful in the hands of a user when the implemented requirements do not reflect the intended use. Quality in use addresses the degree to which a product is fit for purpose when exposed to a particular context of use.

## Quality Models

Measurable elements of software quality, i.e. quality characteristics, have to be defined in order to assess the quality of a software product and to set quality objectives. A series of attempts to define attributes of software products by which quality can be systematically described (see [Mil02]) has been combined in the ISO/IEC standards 9126:2001 [ISO01] and 25000:2005 [ISO05] respectively. The standards provides a quality model with six quality characteristics, namely functionality, reliability, usability, efficiency, maintainability and portability, which are further refined in sub-characteristics (see Figure 10).

Bugs, i.e. defects, indicate the deviation of the actual quantity of a quality characteristic from the expected quantity. Defects are often associated with deviations in the behavior of a software system, affecting its functionality. The quality model, however, makes clear that defects concern all quality characteristics of a software system. Hence, a deviation from a defined runtime performance is therefore as much a defect as a deviation from the expected usability or a flawed computation.

Quality models are a valuable vehicle for systematically eliciting quality requirements and for adopting a quality engineering approach covering all relevant qualities of a software product. For example, in the research project *WebTesting*, a guideline for methodical testing of Web-based applications (see [RWW+02] and [SRA06]) has been derived from a domain-specific quality model.




Characteristics and sub-characteristics of software quality. FIGURE 10

## Quality Assurance Measures

Quality must be built into a software product during development and maintenance. Software quality engineering [Tia05] ensures that the process of incorporating quality into the software is done correctly and adequately, and that the resulting software product meets the defined quality requirements.

The measures applied in engineering of software quality are constructive or analytical in their nature. Constructive measures are technical (e.g., application of adequate programming languages and tool support), organizational (e.g., enactment of standardized procedures and workflows), and personnel measures (e.g., selection and training of personnel) to ensure quality a priori. These measures aim to prevent defects through eliminating the source of the error or blocking erroneous human actions. Analytical measures are used to asses the actual quality of a work product by dynamic checks (e.g., testing and simulation) and static checks (e.g., inspection and review). These measures aim to improve quality through fault detection and removal.

## Economic Perspective on Software Quality

Applying quality assurance measures involves costs. The costs of achieving quality have to be balanced with the benefits expected from software quality, i.e., reduced failure costs and improved productivity. Engineering of software quality, thus, is driven by economic considerations, entailing what Garvin [Gar84] described as "value-based approach" to define quality.

Value-based software engineering [BAB+05] therefore elaborates on the question "How much software quality investment is enough?" [HB06]. In [RBG05] we describe how an analysis of the derived business risks can be used to answer this question when making the investment decision, which can be stated as trade-off. Too little investments in quality assurance measures incur the risk of delivering a defective product that fails to meet the quality expectations of customers and results in lost sales. This risk has to be opposed with the risk of missed market opportunities and, thus, lost sales due to too much quality investments prolonging the time-to-market. Neither too little nor too much quality investments are economically reasonable. From an economic perspective a "good enough" approach to software quality [Bac97] is considered the optimal solution.

Engineering of software quality in practice has to be coherent with economic constraints. Hence, in any application-oriented research, the economic perspective of software quality is a dominant factor. Further examples about economic considerations will be presented in the next subsections as part of the discussion about manual versus automated testing and the prioritization of tests based on the prediction of defect-prone software modules.

## 3.2 Management and Automation of Software Testing

Software testing is one of the most important and most widely practiced measures of software quality engineering [LRFL07] used to validate that customers have specified the right software solution and to verify that developers have built the solution right. It is a natural approach to understand a software system's behavior by executing representative scenarios within the intended context of use with the aim to gather information about the software system. More specifically, software testing means executing a software system with defined input and observing the produced output, which is compared with the expected output to determine pass or fail of the test. Accordingly, the IEEE Standard 610.12-1990 defines testing as "the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component" [IEE90].

Compared to other approaches to engineer software quality, testing provides several advantages, such as the relative ease with which many of the testing activities can be performed, the possibility to execute the program in its expected environment, the direct link of failed tests to the underlying defect, or that testing reduces the risk of failures of the software system. In contrast, however, software testing is a costly measure due to the large number of execution scenarios required to gather a representative sample of the real-world usage of the software system. In fact, the total number of possible execution scenarios for any non-trivial software system is so high that

complete testing is considered practically impossible [KFN99]. Test design techniques (e.g., [Bei90, Cop04]) are therefore applied to systematically construct a minimal set of test cases covering a representative fraction of all execution scenarios. Still, testing can consume up to 50 percent and more of the cost of software development [HB06].

As a consequence, automation has been proposed as a response to the costly and labor-intensive manual activities in software testing. Test automation [FG99] has many faces and concerns a broad variety of aspects of software testing: The automated execution of tests, the automated setup of the test environment, the automated recording or generation of tests, the automation of administrative tasks in testing. In all these cases, tool support promises to reduce the costs of testing and to speed up the test process.

In the following, we present results from research projects conducted at SCCH that involved tool-based solutions addressing different aspects of test automation.

- The first example, *TEMPPO*, outlines the tool support for managing large test case portfolios and related artifacts such as test data, test results and execution protocols.
- In the second example, a framework for the automation of unit tests in embedded software development has been used to introduce the paradigm of test-driven development to a large software project in this domain.
- We conclude this subsection with a study about balancing manual and automated software testing subsuming ongoing observations and lessons learned from several research and industrial projects. In addition, we present a tool-based approach (*TestSheets*) for user interface testing as an example for blending automated and manual testing.

### Tool Support for Test Management

Testing tools are frequently associated with tools for automating the execution of test cases. Test execution, however, is only one activity in the software testing process, which also involves test planning, test analysis and design, test implementation, evaluating exit criteria and reporting, plus the parallel activity of test management. All of these activities are amenable to automation and benefit from tool support.

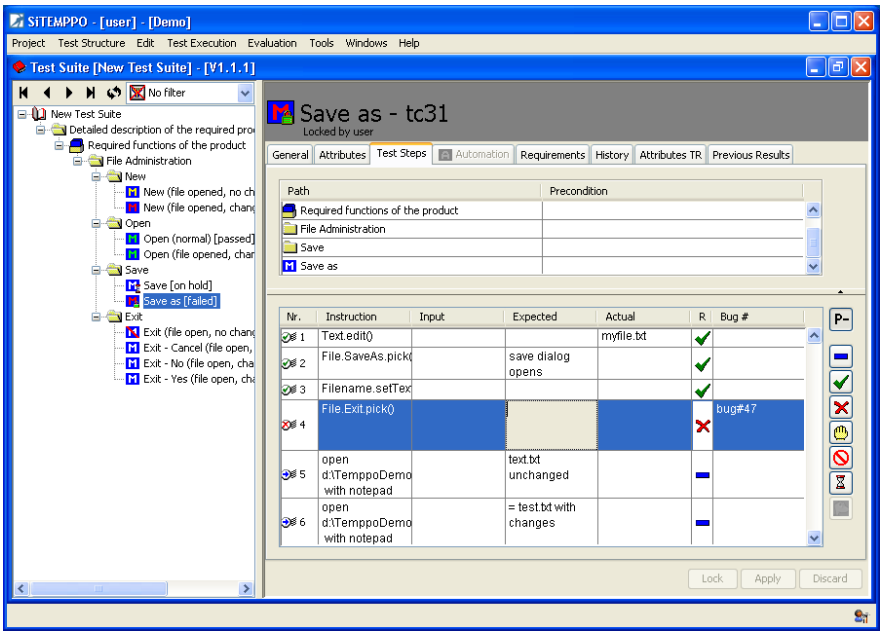
In the following we describe a tool-based approach specifically for test management and present some results from the research project TEMPPO (Test Execution Managing Planning and rePorting Organizer) conducted by Siemens Austria and SCCH. The project results are an excellent example for the sustaining benefit that can be achieved by linking science and industry. The project fostered a fruitful knowledge exchange in both directions. Requirements for managing testing in step with actual practice in large soft-

ware development projects have been elicited by Siemens, and appropriate solution concepts have been developed by researchers at SCCH. The cooperation led to a prototype implementation of a test management environment that addressed a number of research issues significant for tool-based test management in industrial projects.

- A light-weight test process for managing the different stages in the genesis of test cases had to be defined, providing support for the inception of the initial test ideas based on a software requirements specification, the design of test cases and their implementation, the manual test execution as well as the automated execution in subsequent regression testing.
- An efficient structure for organizing and maintaining large hierarchical portfolios of up to several thousand test cases had to be developed. The high volumes of related data included an extendable set of meta-information associated to test cases and a range of artifacts such as associated test scripts, test results and execution protocols accumulated over the whole software development and maintenance lifecycle.
- Changes of the test structure and test cases are inevitable in any large software project once new requirements emerge or test strategies are updated. To accommodate these changes, an integrated versioning and branching mechanism became necessary. It makes sure that results from test executions are linked to the executed version of the test cases even after changes took place.
- Sophisticated query and grouping aids had to be applied for constructing test suites combining a set of test cases for execution. Results from several consecutive test executions had to be merged in a coherent test report for assessing and analyzing the project's quality status.
- Test management as the coordinating function of software testing interacts with a variety of other development and testing activities such as requirements management and change and defect management. For example, the integration of test management and unit testing is described in [RCS03]. These integrations imply interfaces that realize a synchronization between the underlying concepts and workflows of test management and the intersecting activities, which go beyond a mere data exchange between the involved tools.

The prototype developed in the joint research project has been extended with additional features by Siemens and evolved to an industry-strength test management solution. SiTEMPPO[7] (Figure 11) has been successfully applied in projects within the Siemens corporation all over the world, and it is licensed as commercial product for test management on the open market with customers from a broad range of industrial sectors and application domains.

[7] http://www.pse.siemens.at/SiTEMPPO




The test management solution SiTEMPPO. FIGURE 11

## Automation of Unit Testing in Embedded Software Development

Test-driven development (TDD) [Bec02] has been one of the outstanding innovations over the last years in the field of software testing. In short, the premise behind TDD is that software is developed in small increments following a test-develop-refactor cycle also known as red-green-refactor pattern [Bec02].

In the first step (test), tests are implemented that specify the expected behavior before any code is written. Naturally, as the software to be tested does not yet exist, these tests fail – often visualized by a red progress bar. Thereby, however, the tests constitute a set of precisely measurable objectives for the development of the code in the next step. In the second step (develop), the goal is to write the code necessary to make the tests pass – visualized by a green progress bar. Only as much code as necessary to make the bar turn from red to green should be written and as quickly as possible. Even the intended design of the software system may be violated if necessary. In the third step (refactor), any problematic code constructs, design violations, and duplicate code blocks are refactored. Thereby, the code changes performed in the course of refactoring are safeguarded by the existing tests. As soon as change introduces a defect breaking the achieved behavior, a test will fail

and indicate the defect. After the refactoring has been completed, the cycle is repeated until all planned requirements have finally been implemented.

Amplified by the paradigm shift towards agile processes and the inception of extreme programming [BA04], TDD has literally infected the developers with unit testing [BG00]. This breakthrough is also attributed to the framework JUnit[8], the reference implementation of the xUnit family [Ham04] in Java. The framework provides the basic functionality to swiftly implement unit tests in the same programming language as the tested code, to combine related tests to test suites, and to easily run the tests or test suites from the development environment including a visualization of the test results.

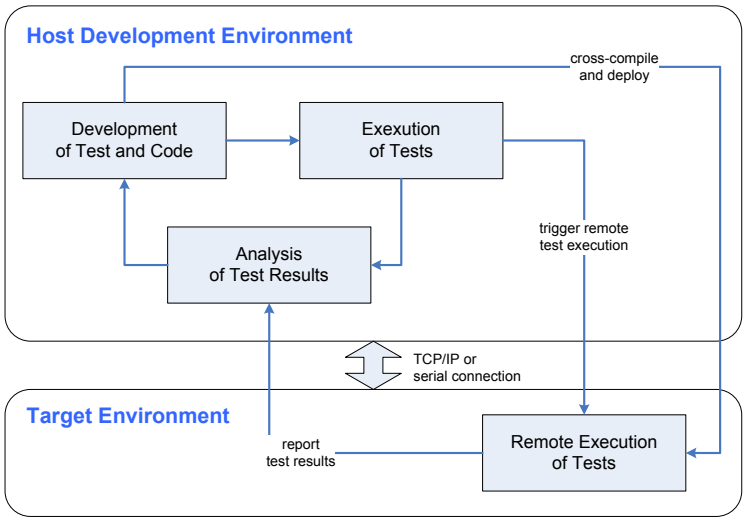
TDD has been successfully applied in the development of server and desktop applications, e.g., business software or Web-based systems. The development of embedded software systems would also benefit from TDD [Gre07]. However, it has not been widely used in this domain due to a number of unique challenges making automated unit testing of embedded software systems difficult at least.

- Typical programming languages employed in embedded software development have been designed for runtime and memory efficiency and, thus, show limited support for writing testable code. Examples are limitations in error and exception handling, lack of comprehensive meta-information, rigid binding at compile-time, and little encouragement to clearly separate interfaces and implementation.
- The limiting factor is usually the underlying hardware with its harsh resource and timing constraints that forces the developers to design for runtime and memory efficiency instead for testability. When the code is tuned to produce the smallest possible memory footprint, debugging aids as well as additional interfaces to control and to introspect the state of the software system are intentionally removed.
- Cross-platform development with a separation between host development environments and target execution platforms is a typical approach in building embedded software systems. The development tools run in a host environment, usually including a hardware simulator. Larger increments are cross-compiled and tested on the actual target system once it becomes available.
- In addition, unit testing is concerned with a number of domain-specific issues causing defects that demand domain-specific test methods and tool support. In embedded software development, these specific issues include, for example, real-time requirements, timing problems, and asynchronous execution due to multi-threaded code or decentralized systems.

The goal of the project was to tackle these challenges and to introduce the concept of TDD to the development of embedded software for mobile and handheld devices. Together with the partner company we developed a framework for automated unit testing with the aim to resemble the design of the

[8] http://www.junit.org

xUnit family as closely as possible, so unit tests could be written in the restricted C++ language variant used for programming embedded devices. Beyond that, the framework comprises extensions such as to run as application directly on the mobile device or to remotely execute unit tests on the target device via a TCP/IP or a serial connection, while the test results are reported back to the the development environment on the host (Figure 12). Many defects only prevalent on the target hardware can so be detected early in development, before the system integration phase.




Workflow for unit testing in the host development environment as well as on the target device. FIGURE 12

## Balancing Manual and Automated Software Testing

Questions like "When should a test be automated?" or "Does test automation make sense in a specific situation?" fuel an ongoing debate among researchers and practitioners (e.g. [BWK05]). Economic considerations about automation in software testing led to the conclusion that – due to generally limited budget and resources available for testing – a trade-off between manual and automated testing exists [RW06]. An investment in automating a test

reduces the limited budget and, thus, the number of affordable manual tests. The overly simplistic cost models for automated testing frequently found in the literature tend to neglect this trade-off and fail to provide the necessary guidance in selecting an optimally balanced testing strategy taking the value contribution of testing into account [Ram04].

The problem is made worse by the fact that manual and automated testing cannot be simply traded against each other based on pure cost considerations. Manual testing and automated testing have largely different defect detection capabilities in terms of what types of defects they are able to reveal. Automated testing targets regression problems, i.e. defects in modified but previously working functionality, while manual testing is suitable for exploring new ways in how to break (new) functionality. Hence, for effective manual testing detailed knowledge about the tested software system and experience in exploring a software system with the aim to find defects play an important role [BR08]. In [RW06] we propose an economic model for balancing manual and automated software testing and we describe influence factors to facilitate comprehension and discussion necessary to define a value-based testing strategy.

Frequently, technical constraints influence the feasibility of automaton approaches in software testing. In the project *Aragon*, a visual GUI editor as a part of an integrated development environment for mobile and multimedia devices, has been developed [PPRL07]. Testing the highly interactive graphical user interface of the editor, which comprises slightly more than 50 percent of the application's total code, involved a number challenges inherent in testing graphical user interfaces such as specifying exactly what the expected results are, testing of the aesthetic appearance, or coping with frequent changes.

While we found a manual, exploratory approach the preferable way of testing the GUI, we also identified a broad range of different tasks that can effectively be automated. As a consequence we set up the initiative *TestSheets* utilizing Eclipse cheat sheets for implementing partial automated test plans embedded directly in the runtime environment of the tested product [PR08]. This integration enabled active elements in test plans to access the product under test, e.g., for setting up the test environment, and allows to tap into the product's log output. Test plans were managed and deployed together with the product under test.

We found that partial test automation is an effective way to blend manual and automated testing amplifying the benefit of each approach. It is primarily targeted at cumbersome and error-prone tasks like setting up the test environment or collecting test results. Thereby, partial automation enhances the capability of human testers, first, because it reduces the amount of low-level routine work and, second, because it provides room for exploring the product under test from various viewpoints including aspects like usability, attractiveness and responsiveness, which are typically weakly addressed by automated tests.

## Monitoring and Predicting Software Quality 3.3

Software quality engineering is an ongoing activity. Beyond measures to achieve software quality, it requires paying close attention to monitor the current quality status of software systems and to anticipate future states as these software systems continue to evolve. In the following we show how a research project integrating software engineering data in a software cockpit can provide the basis for monitoring and predicting software quality of upcoming versions of software products.

### Software Cockpits

Continuous monitoring and management of software quality throughout the evolution of a software system [MD08] requires a comprehensive overview of the development status and means to drill-down on suspicious details to analyze and understand the underlying root causes. Software cockpits (also known as dashboards or software project control centers [MH04]) have been proposed as key to achieve this vision by integrating, visualizing and exploring measurement data from different perspectives and at various levels of detail. Typical sources of measurement data are software repositories and corporate databases such as versioning systems, static code and design analysis tools, test management solutions, issue tracking systems, build systems, and project documentation.

Each of these repositories and databases serves a specific purpose and provides a unique view on the project. For a holistic view on software quality, the relevant aspects of these individual views have to be integrated. Thereby, in order to support the analysis of the project situation, it is not enough to simply present the data from different sources side by side. The integration requires modeling and establishing the relationships between the different software repositories and databases at data level [RW08]. The topic of data integration has been successfully addressed by the concept of data warehouses with its associated ETL (extract, transform, load) technologies in database research and practice [KC04].

Data warehouses are the basis for business intelligence solutions, which support managers in making decisions in a dynamic, time-driven environment based on information from diverse data sources across an organization. Test managers and quality engineers operate in a similar environment under pressure to meet high-quality standards and, at the same time, to deliver in a tight schedule and budget. Hence, as partner in the competence network

*Softnet Austria*[9] we investigated and adopted the idea of business intelligence for software development and quality engineering [LR07].

In a study of existing approaches and solutions offering software cockpits for testing and quality management, we found an overemphasis of the reporting aspect. The main purpose of most of the studied cockpits was to generate static views of aggregated data, usually retrieved from a single data source. In contrast, Eckerson [Eck05] illustrates the nature of cockpits as the intersection between static reporting and interactive analysis. We therefore implemented a software cockpit with the objective to further explore the requirements and solution concepts for interactive data analysis. We based the cockpit on an open source data warehouse as platform for integrating project-specific data sources from development and test tools. The retrieved data was harnessed in customized software metrics and models [Kan02], which were visualized and analyzed via the cockpit. Our first prototype implementation of the software cockpit supported data extraction from open source software engineering tools such as the issue tracking tool Bugzilla or the versioning system CVS.

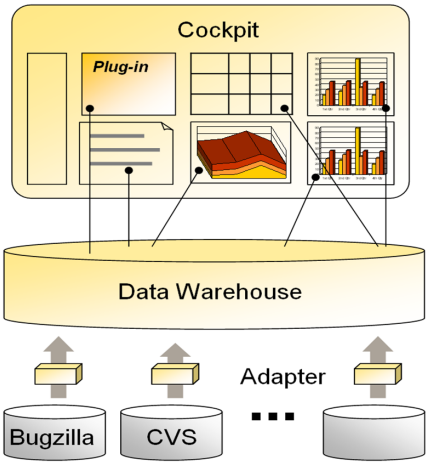
The three main tiers of the cockpit's architecture are shown in Figure 13 (from bottom to top):

1. *Data adapters* periodically extract relevant data from different repositories and databases, e.g., Bugzilla's issue database or the change log of CVS. The data is transformed to a standard data structure and stored in the central data warehouse.
2. *The data warehouse* organizes the data as cubes amenable for on-line analytical data processing. The data schema supports recording the project history for analyzing the evolution and forecasting of trends.
3. *The user interface* of the cockpit visualizes aggregated information and offers the flexibility to customize views, metrics and models. The Web-based implementation provides easy access to visual representation of the integrated data.

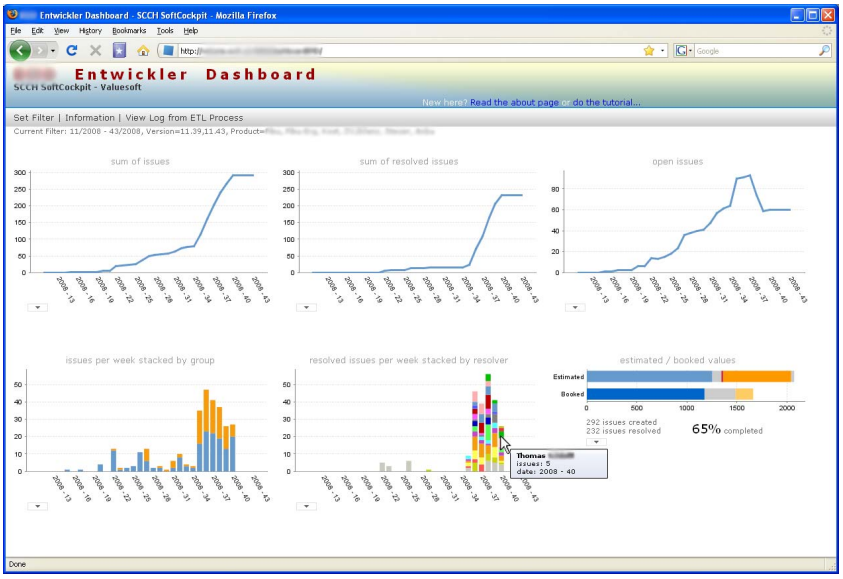
The first prototype of the cockpit has been developed in close cooperation with an industrial software project pursuing an iterative development process. Over a series of rapid prototyping cycles, the Web-based user interface (Figure 14) has evolved including a number of features to visualize and to analyze quality-related measurement data. Building on these results, the software cockpit has been successfully adopted in other projects and organizations, for example, a software product company developing business software involving a development team of more than 100 persons [LRB09].

We identified a number of features that constitute key success factors for the successful implementation and application of software cockpits in practice.

[9] http://www.soft-net.at/




System architecture of the software cockpit. FIGURE 13




Interface of the software cockpit for developers. FIGURE 14

- A user-centered design that supports the users' daily activities keeps the administrative overhead at a minimum and is in line with personal needs for feedback and transparency.
- A comprehensive overview of all relevant information is presented as a set of simple graphics on a single screen as the lynchpin of the software cockpit. It can be personalized in terms of user specific views and filters.
- The presented information (i.e. in-process metrics from software development and quality engineering) is easy to interpret and can be traced back to the individual activities in software development. Abstract metrics and high-level indicators have been avoided. This encourages the users to reflect on how their work affects the overall performance of the project and the quality status of the product.
- In addition, the interactive analysis of the measurement data allows drilling down from aggregated measurements to individual data records and in-place exploration is supported by mechanisms such as stacked charting of data along different dimensions, tooltips showing details about the data points, and filters to zoom in on the most recent information.

## Predicting Defect-prone Modules of a Software System

Data about the points in time where defects are introduced, reported, and resolved, i.e. the lifecycle of defects [Ram08], is gathered in the data warehouse and can be used to construct the history and current state of defective modules of a software system. The data about the software system's past states can also serve as the basis for predicting future states of a software system, indicating which modules are likely to contain defects in upcoming versions.

The rationale for identifying defect-prone modules prior to analytical quality assurance (QA) measures such as inspection or testing has been summarized by Nagappan et al.: "During software production, software quality assurance consumes a considerable effort. To raise the effectiveness and efficiency of this effort, it is wise to direct it to those which need it most. We therefore need to identify those pieces of software which are the most likely to fail—and therefore require most of our attention." [NBZ06] As the time and effort for applying software quality assurance measures is usually limited due to economic constraints and as complete testing is considered impossible for any non-trivial software system [KFN99], the information about which modules are defect-prone can be a valuable aid for defining a focused test and quality engineering strategy.

The feasibility and practical value of defect prediction has been investigated in an empirical study we conducted as part of the research project *Andromeda*, where we applied defect prediction for a large industrial software system [RWS+09]. The studied software system encompasses about 700

KLOC of C++ code in about 160 modules. Before a new version of the system enters the testing phase, up to almost 60 percent of these modules contain defects. Our objective was to classify the modules of a new version as potentially defective or defect-free in order to prioritize the modules for testing. We repeated defect prediction for six consecutive versions of the software system and compared the prediction results with the actual results obtained from system and integration testing.

The defect prediction models [KL05] we used in the study have been based on the data retrieved from previous versions of the software system. For every module of the software system the data included more than 100 metrics like the size and complexity of the module, the number of dependencies to other modules, or the number of changes applied to the module over the last weeks and months. Data mining techniques such as fuzzy logic-based decision trees, neural networks, and support vector machines were used to construct the prediction models. Then, the models were parameterized with the data from the new versions to predict whether a module is defective or defect-free.

Preliminary results showed that our predictions achieve an accuracy of 78 (highest) to 67 percent (lowest). On average 72 percent of the modules were accurately classified. Hence, in case testing has to be stopped early and some modules have to be left untested, a test strategy prioritizing the modules based on the predicted defectiveness is up to 43 percent more effective than a strategy using a random prioritization. Even in with the lowest prediction accuracy the gain can be up to 29 percent compared to a random testing strategy when only 60 percent of all modules are tested. The gain over time is illustrated in Figure 15. The testing strategy based on average defect prediction results (blue) is compared to the hypothetical best case—a strategy ordering the modules to be tested according to their actual defectiveness (green)—and the worst case—a strategy ordering the modules purely random (red).

The depicted improvements in testing achieved by means of defect prediction are intermediate results from ongoing research. So far, the prediction models have been based on simple metrics derived from selected data sources. Combining the data in more sophisticated ways allows including additional aspects of the software system's history and, thus, promises to further increase the prediction performance [MGF07]. In a specific context of a project, the results can be improved even further by tuning of the applied data mining methods. For the future, we plan to extend this work to a larger set of industrial projects of various sizes and from different domains.

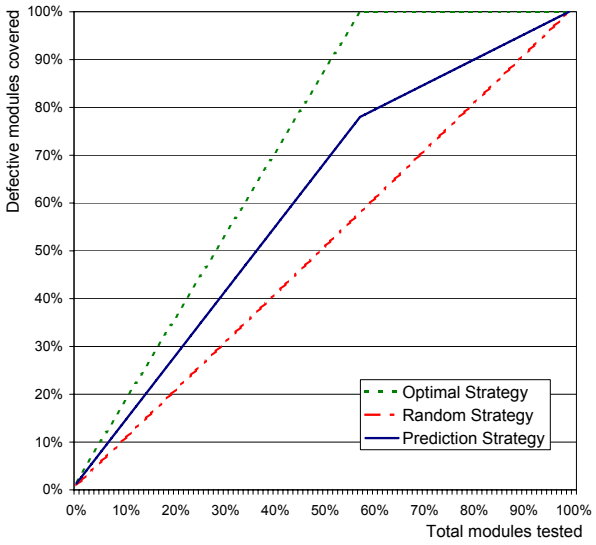



FIGURE 15 Improvement gain achieved by defect prediction.

# 4 Software Architecture Engineering

A software system's architecture is an abstraction of its implementation, omitting details of implementation, algorithm and data representation, see [BCK03]. The architecture of a software system is often represented by different models, each consisting of abstract elements and relationships. Each of these models can be used to describe a particular abstract view of important structural relationships, facilitating understanding and analysis of important qualities of a software system. The fact that the abstraction defined by an architecture is not made up by one but by different structures and views providing different perspectives on a software system is reflected in a widely accepted definition of software architecture provided by the Bass et al. [BCK03], where software architecture is defined as "the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them". Architecture is the result of design. This is reflected in a broader definition provided by Medvidovic et al. [MDT07] which state that "a software system's architecture is the set of principal design decisions about a system". This includes design decisions related to structure, behavior, interaction, non-functional properties, the development process, and to a system's business position (see [MDT07]).

The architecture of a software system is one of the most important concepts during software development. It is among the first artifacts produced in the development and contains *early design decisions* [Cle96] that are usually long lasting and expensive to change later in the development. Fundamental system qualities like performance, security, or scalability are determined by a system's architecture. For developers it is a *blueprint* [Gar00] of the system that guides and constrains implementation. For stakeholders it is a means of communication. Formal architecture representations (architecture models) can be used for automatic analysis of important system qualities; informal architecture representations (architecture documentation) are an important means of communication during and system documentation during all phases of a software life-cycle.

Superficially, software architecture research can be classified into two main research areas: general software architecture research and domain-specific software architecture research. General software architecture research encompasses concepts, methods, and tools for creating, describing, and managing software architectures, ideally independent of a particular domain. Domain-specific software architecture research focuses on the application and adaption of general software architecture principles for specific domains and includes the creation of domain-specific architecture languages, tools, reference architectures, and patterns.

In this section we describe both basic and applied research projects in the area of software architectures that are conducted by SCCH with academic and industrial partners. The remainder of this section is structured as follows: Subsection 4.1 provides an overview of the field of software architectures and highlights important research challenges. Subsection 4.2 describes a research cooperation between SCCH and the Software Engineering Group at the Department of Business Informatics, Johannes Kepler University Linz, focusing on languages and tools for comprehensive and integrated software architecture management. Subsection 4.3 describes applied software architecture research for industrial software systems. Finally, Subsection 4.4 describes architecture-related research activities for enterprise information systems.

## General Research Areas and Challenges 4.1

In this subsection we provide an overview of research fields and challenges in the area of general software architecture research. Important research fields in this area are architecture design, architecture implementation, architecture analysis and evaluation and architecture documentation [TvdH07, KOS06].

### Architecture Design

Architecture design is the activity that creates a system's architecture. While design is performed in all phases of software development fundamental design decisions are usually made before the implementation phase [TMD09]. When creating the initial architecture, a wide range of aspects has to be taken into consideration. Since it is not possible to fulfill the concerns of all stakeholders the architecture of a software system represents a tradeoff between stakeholders concerns [RW05]. Architecture design experience is represented by patterns [MKMG97], styles [MKMG97], and reference architectures [TMD09]. Architecture is also implicitly reused through the use of frameworks [SB03] and middleware [SB03] for a particular domain.

### Architecture Implementation

Implementing a defined architecture is in fact the task of mapping the concepts defined during design to implementation artifacts [TMD09]. This process can be performed manually and automatically. If the architecture is implemented manually, a major problem is to ensure that the system implementation conforms to the intended architecture. This problem is known as architectural decay [AGM05], architectural drift [PW92], architectural erosion [PW92] and design erosion [vGB02]. If existing libraries, middleware and frameworks are used, architecture and design is effectively reused and the possible architectural drift is reduced. Since a framework usually defines the architecture for a family of applications in a particular domain, architectural drift is still possible. Approaches for checking architecture conformance like *Lattix* [SJSJ05], *Sotoarc* [Sof07] and *SonarJ* [Hel07] operate at the level of programming-language concepts. They lack high-level architecture support and are not integrated well enough with the development process. Model-driven development [Sch06] can be used for automatically deriving (parts of) an implementation from an architecture. Since architecture is an abstraction of a software system and no specification, only code skeletons can be generated, which have to be implemented manually. The synchronization of the manually modified code with the models used for code generation is a central problem of model-driven software development [HT06].

### Architecture Analysis and Evaluation

The purpose of software architecture analysis is to analyze the software architecture to identify potential risks and verify that the quality requirements have been addressed in the design [LH93]. Analysis activities can take place

before the system has been build, during it is built and after the system has been built [DN02]. Architecture analysis can be performed manually by using architecture evaluation methods or automatically using architecture analysis tools.

Architecture evaluation methods like the *Software Architecture Analysis Method* (SAAM) [CKK02] or its successor the *Architecture Tradeoff Analysis Method* (ATAM) [CKK02] are scenario-based evaluation methods that have been developed particularly to validate quality attributes, which are usually difficult to analyze. Architecture evaluation methods are time-consuming and resource-intensive processes. They are usually used for evaluating the initial design of a software system with its stakeholders and for assessing the architecture of an already implemented system. They are not intended for continuous architecture analysis.

Architecture Description Languages (ADLs) are formal languages to represent the architecture of a software system [Cle96]. They allow the automatic analysis of system properties before it has been built [Cle95]. An ADL describes a system in terms of components, connectors and their configurations [MT00]. Usually ADLs have a textual as well as a graphical representation [Cle96]. A large number of general purpose and domain-specific ADLs exist [MT00]. Disadvantages of ADLs are lack of tool support [MDT07, MT00], lack of implementation integration [MT00] and lack of standardization. Some ADLs allows code generation in the sense of model-driven software development, which may lead to problems in synchronizing architecture and code as mentioned above. While UML is sometimes discussed as a general purpose ADL, its suitability as an ADL is still subject of study and debate [MDT07]. In most cases UML, is used for architecture documentation as described below.

### Architecture Documentation

Since the architecture of a software system is not entirely contained in the implementation it must be documented separately [Hof05]. Documenting software architecture is quite different from architectural descriptions that are created for analysis [IMP05]. While the latter requires a formal description that can be processed by tools, architecture descriptions for documentation purposes are usually described informal using natural language. Researchers have proposed a view-based approach for describing software architectures [RW05, Kru95, CBB+02, HNS99]. An architectural view is a representation of a system from the perspective of an identified set of architecture-related concerns [Int08]. Architecture documentations usually consist of multiple views. The concepts of view-based architecture documentation are defined in the *ISO/IEC 42010 standard: Systems and Software Engineering – Architectural Description* [Int08].

The *Unified Modeling Language* (UML) [Obj07] is the standard modeling language in software development. UML is primary a visual notation [WH05] that consists of over ten different loosely connected individual notations [MDT07]. UML is often used in architecture documentation. UML 2.0 has adapted many features from ADLs [AGM05].

## 4.2 Software Architecture Management – Languages and Tools

While numerous approaches for individual architecture-related activities exist and have found wide-spread use in practice, architecture is still not supported well enough during software development. Often it is not explicitly and entirely documented; system implementation deviates from the intended architecture; the documented architecture is out-of-date; architecture conformance checking is not performed continuously and can only be performed at a low level of abstraction; and finally the architecture is usually described informal and cannot be used for automatic analysis.

The *Software Architecture Engineering* (SAE) project is a strategic project that addresses these problems. The main idea is to support software architecture related activities like modeling, changing, and validating software architectures as an integral part of other activities in software development. This is achieved by a central integrated and formalized language for describing software architectures (LISA) and an unobtrusive set of integrated architecture tools working on this model (LISA-toolkit).

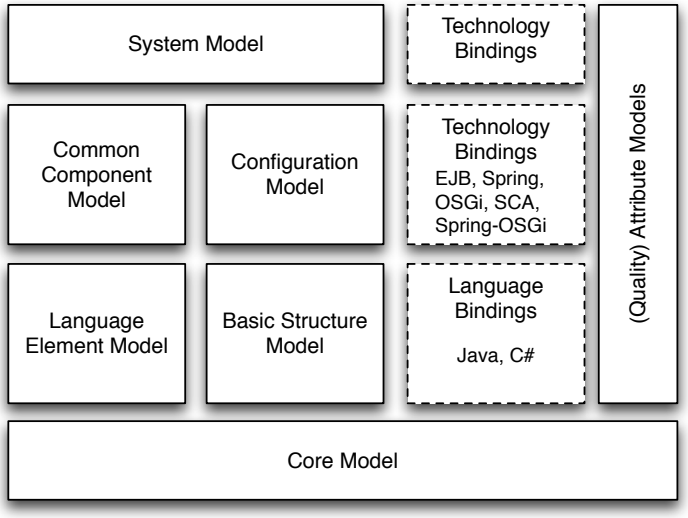
### Integrated Architecture Language

*LISA* is an architecture description language which can be used for describing and validating architectural structures and properties. It has no explicit textual representation, which is intended for manually creating an architectural description. Instead, LISA-based architecture models are manipulated by the tools provided by the LISA toolkit.

While LISA has been designed as a general purpose ADL, it has been designed with specific aims promising to solve some of problems of architecture engineering described above and with the central aim to raise the abstraction in software development further by extending modeling concepts provided by general purpose programming languages by architectural concepts.

From a technical perspective LISA is an extensible meta-model based on XML-Schema and consists of several integrated and connected sub models. Contrary to other ADLs it puts a special emphasis on linking architecture to

implementation and on supporting existing component-models and technologies. The main elements of LISA are shown in Figure 16. The lower layers




The LISA Model. FIGURE 16

of the LISA language definition shown in Figure 16 can be used for describing architectural relationships that are defined statically in code. Model elements at these lower layers can be partly extracted from or mapped to source code. Examples are the elements of the *Language Element Model*, which include concepts like classes and interfaces. These elements can be organized by structures in the *Basic Structure Model*. The *Basic Structure Model* can be used for defining elements like functional units, subsystems, deployment units, and layers. Together the elements of the lower layers of the LISA language definition enable functionality provided by architecture management tools. This includes usage and dependency analysis, synchronizing architecture with code, and defining and checking architectural constraints at the level of programming language concepts. Although the lower layers of LISA are aligned with concepts found in concrete programming languages they are still abstract. Bindings to particular programming languages are provided by *Language Binding* definitions as shown in Figure 16.

The upper layers of LISA include the definition of abstract models for describing components, configurations, and whole systems. Again the binding to specific component technologies and models is provided by *Technology Binding Models*. Currently LISA supports bindings for EJB [EJB06], Spring

[Spr08b], OSGi [OSG07], Spring Dynamic Modules for OSGi [Spr08a], and SCA [SCA07]. Examples for elements at the higher layers of LISA are component, contract, port, composite, application, location and tier. These elements can be used for describing and analyzing architectures of component-based and distributed service-oriented software systems. In such systems architectural relationships are not defined in code but through late composition and configuration. Finally, *(Quality) Attribute Models* as shown in Figure 16 can be used for attaching semantic attributes and policies to architectural elements at all levels of abstraction. Such attributes can be used for annotating and validating non-functional attributes of a software system.

## Pervasive Architecture Toolkit

LISA is not intended to be used directly by means of a textual representation. Instead creation, manipulation, visualization, and validation of LISA-based architectural models are supported by the LISA-toolkit. To provide unobtrusive and integrated support for architecture related activities during the whole development process, the toolkit is implemented as a set of Eclipse plug-ins and designed to be integrated into software analysis and development tools. An overview of the general structure of the toolkit is shown in Figure 17. A screenshot of the LISA-Toolkit is depicted in Figure 18.

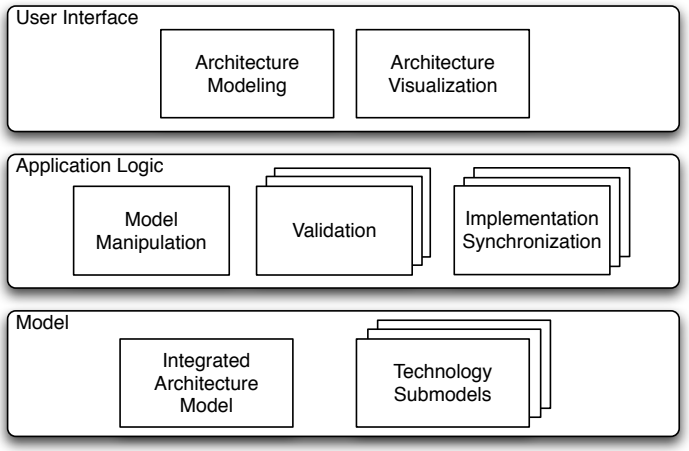
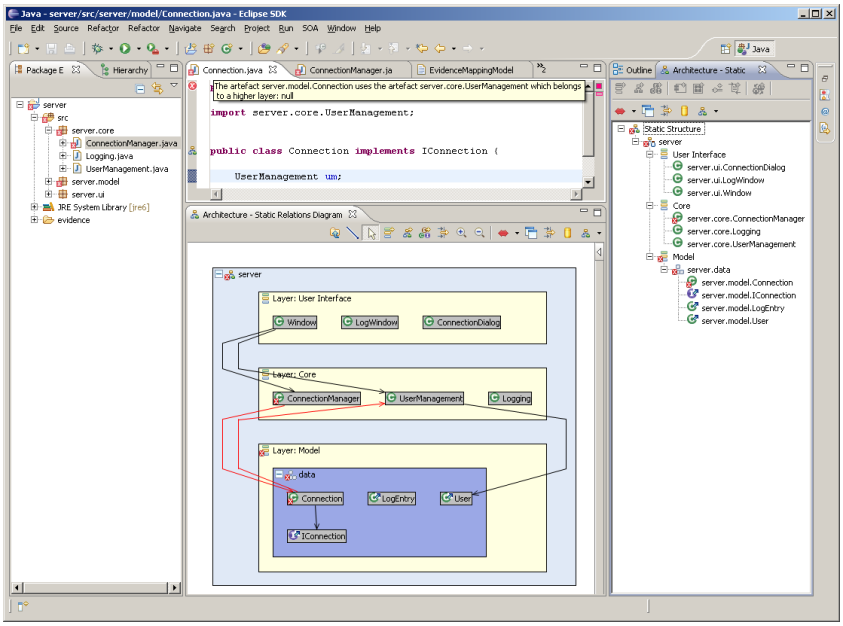



FIGURE 17 The LISA Toolkit.




A Screenshot of the LISA Toolkit. FIGURE 18

As shown in the figure, the toolkit provides an API for editing a LISA-based architecture model as well as functional components for validating architectural constraints and for synchronizing an architecture with a system implementation. In addition, the toolkit provides user interface components for architecture modeling and visualization. All UI components are working on the same architectural model and thus support editing and visualization of different aspects of a system in a consistent way. Violation of architectural constraints defined in the model are immediately shown in all graphical and textual representations of the affected elements.

Examples of available visualizations and modeling tools are shown in Figures 19 and 20. Figure 19 shows usage and dependency relationships of classes and interfaces organized in different layers in an object-oriented software system. The figure shows layer violations (see (1) and (2) in Figure 19) as an example for the violation of architectural constraints.

Figure 20 shows diagrams for representing and editing the architecture of a service-oriented software system using the *Service Component Architecture* (SCA). Instead of classes and interfaces, the main elements at this layer of abstraction are components and contracts. The *Component Decomposition Diagram* provides on overview of the components of the system. In LISA components are independent of a particular implementation technology. The

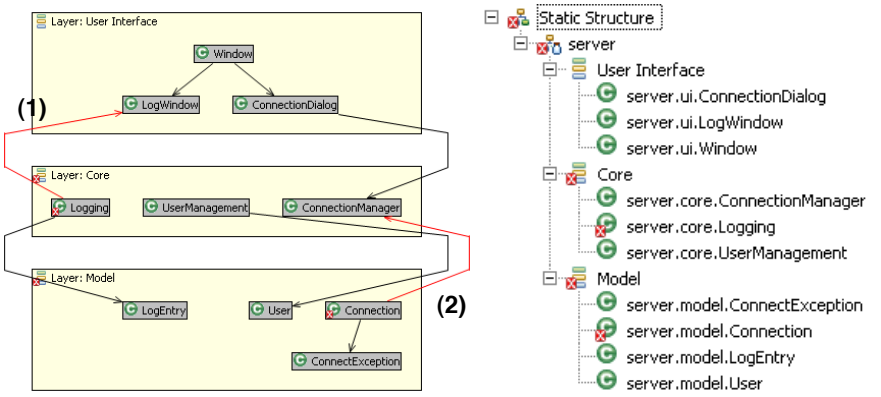



FIGURE 19 Language Element Diagrams.

diagram can be used to explore component structure and decomposition of composite components. Component usage relationships are typically not implemented in source code but rather configured during system assembly or created automatically based on component specifications. The *System Relation Diagram* shows the configuration of a system in terms of components and their relations. Relations and element properties violating architectural constraints are also indicated at this level of abstraction.

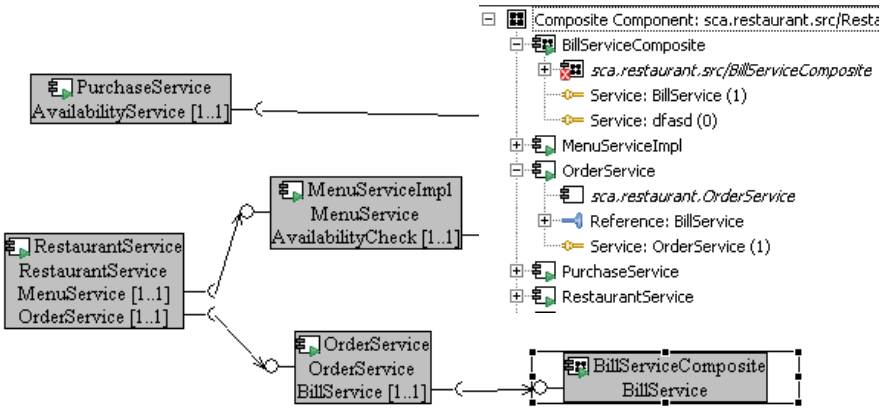



FIGURE 20 Component Diagrams.

### Addressed Problem Domains and Research Challenges

The approach supports architecture related activities in different areas of software development and addresses several challenges described above.

*Architecture Design:* The LISA-toolkit can be used for modeling architectural elements and relationships. A system can be modeled from low level elements at the level of classes and interfaces to higher level components at the level of systems. LISA supports in particular component-based and distributed service-oriented systems.

*Architecture Implementation:* An architecture model can be used for generating component implementation skeletons and other application parts in the sense of model-driven development. However, the central benefit of the approach in this area is the support for binding architecture to implementation and for continuous synchronization of architecture and implementation. This addresses the problems of outdated architecture description and of architectural drift.

*Architecture Analysis and Evaluation:* Since the approach is based on a central formalized architecture model, architectural constraints can be defined similar to other architectural description languages. Contrary to other approaches constraints can be defined at the level of the architecture model as well as the level of the implementation and technology binding. Since the same model is used during the whole software life-cycle, architectural constraints can be checked during analysis, implementation and also during maintenance.

*Architecture Documentation:* Architecture documentation is best supported by a view-based approach as described above. LISA supports architecture documentation by providing diagrams that can be used for describing architectural views.

To summarize, the main elements of the architecture management approach are a central formalized architecture model and set of tools working on this model. Distinct features are the possibility to integrate the architecture tools with other development tools, close integration and synchronization with implementation, the support of component-based systems and the support of existing component models and technologies. First results of the approach have been published in [BW08].

## 4.3 Software Architectures for Industrial Applications

In the context of this subsection, the term *industrial applications* refers to software, which handles tasks related to manufacturing, process control and -automation. This includes applications for manufacturing operations management (e.g. product tracking, product resource management, product data collection) as well as software for basic control of devices and machines. Of course, industrial applications are not solely restricted to manufacturing industries, the term also applies to software solving similar tasks in areas like aviation, the automotive industry, and building automation.

### Challenges and Requirements

Industrial applications tend to have stricter requirements concerning reliability and efficiency (in the sense of ISO-9126) than desktop software. This is particularly the case for software that handles mission- and safety-critical tasks. Reliability means that such industrial applications must provide high availability with minimal downtimes, be robust to failure and be able to fully recover to an operational state in case of faults in underlying or neighbored systems. Efficiency covers the aspects of time and resource behavior. While managing resources like memory efficiently should be important to software applications in general, industrial applications, and especially applications for process control, often must fulfill hard or soft real-time demands. These requirements demand special considerations when designing and implementing industrial applications. Designing real-time systems is particularly challenging [Dou02, Ste00]. Several frameworks aid in designing and developing industrial applications. They range from domain specific control frameworks such as OROCOS [Bru01] to general purpose component frameworks like OSGi [OSG07], which can be used equally for industrial and desktop software.

### Case Study ProMoRTE

*ProMoRTE* [Dor09] is an online (i.e. process controlling) OSGi-based runtime platform for computation algorithms in the steel industry. The following list of requirements was important for designing the architecture of the platform:

- Easy and flexible installation, configuration and operation of computation algorithms at runtime.
- Versioning support for computation algorithms.

- Prescription of unified interfaces for computation algorithms (data access, behaviour control).
- Integration with an IDE for developing and maintaining algorithms.
- Support of legacy code to allow the reuse of existing algorithms coded in C/C++.

The pivotal quality requirement addressed the reliability of the platform. Since it had to be deployed in a 24x7 production environment, a maximum of stability and a minimum of downtime was demanded. Manipulations on a certain algorithm (e.g. reconfiguration), failed computations or crashed algorithm executions had not to influence the operation of other algorithms in the platform. In case of a hardware fault or system crash, a full recovery of the platform and all installed algorithms was mandatory. Other important quality demands concerned portability of the system (OpenVMS, Windows, Linux) and performance. Hard real-time capabilities were not deemed necessary, though.

Of course, these requirements affected the architecture of ProMoRTE. To achieve portability we opted to base our implementation on Java. OSGi as the base framework was chosen for its flexible component oriented architecture allowing hot deployment of components (aka bundles) and providing explicit versioning support. Besides, OSGi easily integrates with the popular Eclipse IDE—since Eclipse builds on OSGi, too.

Support of legacy code and the reliability requirement interfered with each other. Incorporating native libraries in Java applications is done via the Java Native Interface. However, direct integration bears the risk of reduced stability and robustness, because defects in native code can not always be handled by the JNI code or the Java application code. Thus, we had to consider a distributed architecture to satisfy the reliability needs, which is shown in Figure 21.

The distribution of process algorithms had in turn an effect on performance, since communication of platform and algorithms had to pass process boundaries. In the end, the negative effects could be coped with. The platform has been operative in the project partner's productive environment for over two years.

## 4.4 Software Architectures for Enterprise Information Systems

To stay competitive, enterprises need a flexible applications architecture that permits changes and the quick deployment of new functionality with minimal integration effort. Enterprise applications can be integrated at different levels: data, business logic, and presentation. Integration of heterogeneous

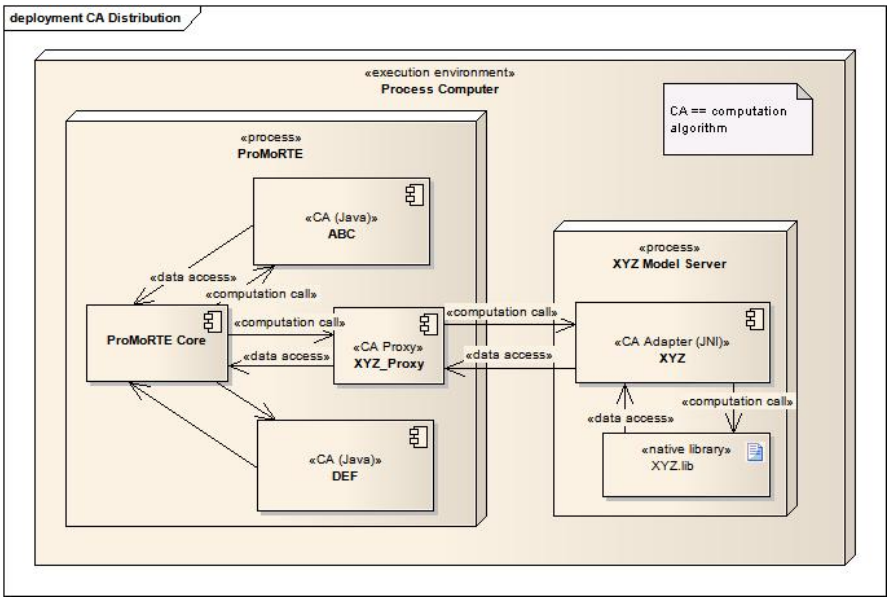



FIGURE 21 Distribution of algorithms with ProMoRTE.

systems both within enterprises (EAI) and between enterprises (B2B) requires standardization. Standardization is a strong trend at all integration levels. Examples are Web Service standards like SOAP and WSDL as well as higher-level standards for B2B-integration like ebXML and RosettaNet.

To increase reusability and to flexibly adapt to changing business conditions and processes, enterprise applications are increasingly decomposed into small reusable and composable elements using standardized interfaces. At the presentation level such elements are portal components, which can be composed to web portals and customizable workplaces. At the business logic layer, the central elements for composition are services. Currently, the term Service-Oriented Architecture (SOA) is usually used for flexible enterprise information system architectures based on services using standardized (Web Service) protocols.

## Challenges and Requirements

Central challenges of integrating components at the presentation level are the integration of the components' user interfaces into a single consistent aggregated application user interface and the support for data exchange between

these components. Data exchange at the presentation level is complicated by two main issues. Presentation level components that are to be integrated may not only be developed by different parties but also managed and operated by different providers. This means that data exchange needs to support remote communication and may cross several different security boundaries. In addition, standardized composition models are needed for integrating components without additional development effort. These issues have been addressed by the Enterprise Portal Project at SCCH, which has been conducted to created customizable workplace solutions in the financial domain.

Central challenges at the business logic level, i.e., for an Service-Oriented Architecture, are the support for service evolution, service reuse, and service management. Similar to the presentation level, services at the business-logic level may be produced and operated by different providers. This means that any changes to the architecture of a SOA-based system may potentially affect not only other departments but even other companies. Equally changes of services operated by a particular company may affect the whole enterprise information architecture. Aspects of service evolution and management in an SOA have been addressed by the *IT4S* project at SCCH.

### Enterprise Portal Project (Enipa)

The main result of the *Enipa* project is a component model for enhanced integration of portal components in web portals. The model supports not only the aggregation of components within one web page, but also the composition of component navigation into a central navigation area, the communication between local and remote components, and heterogeneous environments. The approach is based on existing standards like Portlets and WSRP and uses XML for describing component navigation and communication capabilities. It is declarative and may also be used for improving integration capabilities of already existing portal components (see [WZ05] and [WWZ07]).

### SOA Evolution and Management (IT4S)

The results of the *IT4S* project are an approach for SOA governance and a versioning approach for service evolution. Notable aspects of the governance approach are an extensible model for describing service metadata of arbitrary service types (not only Web services), support for the process of service specification and service creation, a service browser for service reuse, and the support for service evolution through information about service versioning, service dependencies and service installations [DW06]. The versioning

approach consists of a versioning model, of suggestions for release management, evolution scenarios, and a versioning scheme for enterprise services [WZD07]. It also includes compatibility rules for the homogeneous evolution of heterogeneous services [KWZ09].

# 5 Domain-Specific Languages and Modeling

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [vDKV00]. There is a common perception to distinguish between horizontal and vertical domains. Horizontal domains cover different technical software areas as user interface, database, testing, etc. whereas vertical domains are found in business areas like insurance, telephony, or process automation.

Domain-specific languages play an important role in various software engineering methodologies. In generative software development [CEC00], a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages [Cza05]. Domain-specific languages have also been applied for end-user programming [MKB06] [PHS+08b]. There has also been a trend in model-driven development towards representing models using appropriate DSLs resulting in the domain-specific modeling discipline [VS06, KT08].

Domain-specific modeling (DSM) proposes to model a software system and to fully generate code from these high-level models. Importantly, both the language used for modeling as well as code generators are specific to a certain domain, in contrast to other model-driven approaches like model-driven architecture [KWB03] that proposes standard modeling languages, e.g. UML.

Domain-specific modeling is about finding concepts from a certain domain and to specify a domain-specific language from these concepts. Tolvanen and Kelly [TK05] identified following driving factors for language construct identification based on an evaluation of 23 case studies:

1. domain expert concepts,
2. developer concepts,
3. generation output,
4. look and feel of the system build, and
5. variability space,

where the combination of the latter two promises most benefits for DSM solutions [TK05, KT08].

The key contribution of domain-specific languages and modeling is to significantly increase both productivity and quality in software development by raising the abstraction level from general-purpose programming languages and modeling notations toward bespoke domains.

For many realistic industrial software systems, a single DSL only solves a limited part of a problem [KT08, LH09]. An entire system, hence, is typically build from a single or multiple DSLs together with system parts developed traditionally by means of general-purpose programming languages.

The Software Competence Center Hagenberg (SCCH) has carried out various application-oriented research projects together with partner companies related to concepts and techniques of domain-specific modeling and languages:

*Aragon:* development of a graphical-user interface builder for an object-oriented application framework for mobile devices [PPRL07].

*Testbed:* development of a software testbed for mobile software frameworks [HHK+08].

*OdcEditor:* development of an end-user programming environment for injection molding machines [PP09].

The remainder of this section focuses on an overview of the field of domain-specific languages and modeling in general and its application to the aforementioned research projects at SCCH.

## Overview of the Field 5.1

Domain-specific modeling has two mutual dependent goals [KT08]: First, raise the level of abstraction beyond programming by specifying the solution in a language that directly uses concepts and rules from a specific problem domain. Second, generate final products in a chosen programming language or other form from these high-level specifications.

To achieve these goals, a domain-specific modeling solution consists of following parts (see [KT08]): a domain-specific language, code generators, and a domain framework.

### Domain-Specific Language

A domain-specific language (DSL) provides concepts and rules to represent elements in the problem domain on language level. In that, it allows expressing problems of the given domain in a more natural and intuitive way, raises

the level of abstraction in the given domain, and brings software specification closer to the domain experts. In distinction to general-purpose programming languages, which are universally applicable to many domains, domain-specific languages are created specifically for problems in the domain and are not intended to problems outside it.

As a formal language, a DSL is defined by its concrete syntax, abstract syntax, and semantics. The concrete syntax (or notation) specifies the concrete appearance of a DSL visible to the user. The notation can be one of various forms—textual, graphical, tabular, etc.—depending on the problem domain at hand. The concrete syntax usually is of increased importance for DSLs, as it—to a great extent—determines acceptance by users.

The goal of the abstract syntax (or meta-model in context of model-driven development) is to describe the structural essence of a language including elements and relationships between elements like containment and references. Concrete and abstract syntax of textual languages are often defined in a single source [KRV07, GBU08] or the concrete syntax defines the abstract syntax implicitly [LJJ07, PP08].

Whereas the formal definition of both abstract and concrete syntax is well elaborated, the language semantics is usually given by the code generators. Formal definition of language semantics is still an open research field, e.g. [Sad08].

In general, DSLs are either standalone, embedded into a host language, or used as domain-specific presentation extensions. A standalone DSL provides full abstraction of the underlying general-purpose programming language used for the solution [KT08]. An embedded DSL is one which extends an existing general-purpose language (e.g. [AMS08]). A domain-specific presentation extension of a general-purpose programming language (e.g. [EK07]) facilitates readability and closes the gap between problem and solution domain. In context of domain-specific modeling, the focus is on standalone DSLs.

## Code Generators

A code generator extracts information from a DSL program and generates code in a target language. The target code is either in a general-purpose programming language, which then will be compiled and linked with the domain framework [KT08], or is in some intermediate representation that is interpreted by the domain framework [HHK+08].

Code generation works with transformation rules which specify how the different elements in the DSL program are transformed into target language constructs. For textual output, the transformation techniques model-to-text and text-to-text [LJJ07] are used, depending on the representation of the

DSL program (model or text). For the former one, two main approaches are available [CH03]: visitor-based and template-based.

### Domain Framework

A domain framework provides the interface between the generated code and the underlying target platform. Domain-specific frameworks [FJ99] are not specific to the DSM approach but a general approach for software reuse to increase productivity in a specific domain. However, a domain framework can support a DSM approach by providing the immutable part of a solution not visible to users which can be customized and configured by DSL programs. In general, a domain framework is written in a general-purpose programming language by and for software experts, whereas a DSM solution puts a domain-specific language on top of a framework.

### Tool Support

The success of a DSM solution largely depends on provided tool support to create and manipulate software models in a given domain-specific language [PP08]. Building a DSM solution should be possible without having to manually program the tool support.

Figure 22 shows the tool chain supporting a DSM approach. A powerful meta-modeling approach is crucial for defining concrete and abstract syntax of a DSL language. The goal is to generate editors, compilers and other language tools from the declarative specification. On the other side, code generation is facilitated by the specification of code generation rules, which specify how the language elements defined in the meta-model should be translated into target code. Today, several powerful DSL frameworks exist which support language engineering for graphical as well as textual DSLs, e.g., EMF and GMF tools for Eclipse [SBPM09], DSL Tools in MS Visual Studio [CJKW07], or the MetaEdit+ environment [KT08]. A comparison of available platforms for textual languages can be found in [PP08].

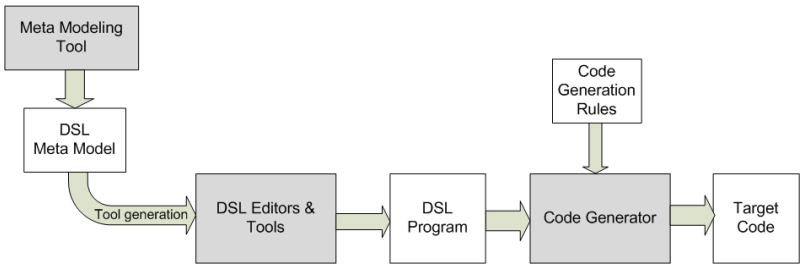



FIGURE 22 DSM tool chain.

## 5.2 Modeling and Code Generation

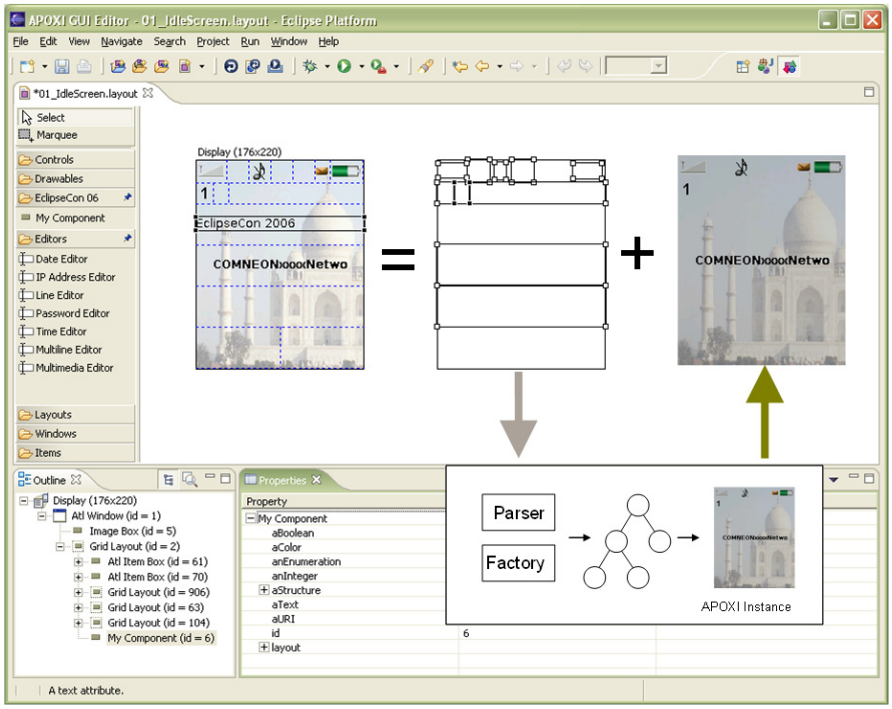
In this subsection, we present results from an application-oriented research project carried out by SCCH and its partner company Comneon[10]. The goal of this project was to develop a platform-independent builder tool for man-machine interfaces (MMI) for mobile and embedded devices. Tool support for graphical user interfaces (GUI) or MMIs are recognized as traditional domains where domain-specific modeling can improve both productivity and flexibility by removing the gap between source code and the resulting MMI. In this research project we have developed modeling, meta-modeling, and code generation concepts and have implemented them in the builder tool Aragon [PPRL07]. The tool is used together with the object-oriented C++ framework APOXI developed by Comneon for supporting the multi-stage customization process carried out by domain experts.

As a typical domain framework, APOXI has been designed to meet the special requirements of mobile, embedded devices and provides specific, easy to use and comprehensible APIs for MMI and application development. Application development follows the general idea of application frameworks [Joh99], combined with a strict separation of application behavior and user interface to ease the MMI customization process. Mobile phone manufacturers (MPM) develop customized software solutions based on the framework and customize and adapt their solutions to meet requirements of different mobile phone network operations and market needs resulting in up to 60 different variants of an MMI. MPM developers are typical domain experts that find it difficult to work with plain C++ APIs provided by APOXI. Hence, there is a need for a domain-specific modeling tool to create, adapt, and customize MMIs of embedded devices.

The main challenge for developing a platform-independent MMI builder tool in the domain of mobile and embedded devices were to provide accurate

[10] www.comneon.com

feedback of a resulting MMI even in the modeling (design) phase for a large set of different devices and to provide code generators that transforms MMI models into platform-specific code that can be used by the APOXI framework. Figure 23 shows a screen dump of the tool Aragon which has been developed on top of the Eclipse platform.




Screen dump of the APOXI GUI Editor. FIGURE 23

## Domain Modeling

Domain modeling in the project started by considering the framework code, i.e., the APOXI framework, the look and feel of resulting MMIs, the requirements of the multi-stage configuration process, and constraints and preferences stemming from the different stakeholders involved. In general, for building MMI solutions of mobile devices a domain expert has to have means to specify the screens with user interface elements, their individual settings,

their layout and positioning, containment relations, and how the user interface gestures are connected to application behavior. In distinction to other UI builder tools, Aragon pursues an approach which is target agnostic, i.e., the tool itself is not dependent on the target implementation but fully configured by meta-information which is realized in a target independent and extensible form.

As result, the meta-model, i.e., the abstract syntax of the modeling language, comprises the following concepts:

- Meta-information on UI components provided by a framework (e.g. APOXI) and extensions like windows and menus together with their attributes, and constraints on their configuration, composition, and layout.
- Meta-information on available applications and features provided by applications. This information is required to connect MMI elements with application features implemented in a general-purpose programming language, e.g., the C++ programming language.

Although, this information can be extracted from source code to a large extend, domain experts want to define further constraints concerning composition of UI components that have to be validated and, hence, must be included in the meta-model. Aragon therefore supports a dual approach, i.e., parts of the meta-information is extracted from source code while additional information can be provided by the modeller. However, the meta-model is the sole information Aragon uses for configuring the editor, filling the component palette with elements, guiding the user actions, and validating domain models (e.g. window composition).

In a typical DSM solution, only a few domain experts define the meta-model, whereas users of a DSM solution are not concerned with it. For Aragon, this is not sufficient because of the multi-stage development process including different stakeholders. Besides the framework team that specify meta-model for the APOXI framework and basic applications, also MPM developers define meta-models about new applications and extensions to the APOXI framework. Hence, more powerful tool support to create, maintain, and validate meta-models by users of the DSM solution is required and provided by Aragon resulting in more flexibility compared to other meta-modeling environments [SBPM09, CJKW07, LKT04]. In Aragon therefore, a flexible meta-modeling scheme based on XML is defined which allows adding meta-information by simply adding additional XML files.

For graphical DSLs, the concrete syntax, i.e. the visual representation of the models, is usually defined by specifying the visual representation of the language elements in the meta-model. In Aragon however, the visual representation is not part of the tool, but visual feedback is provided directly by the target framework, e.g. APOXI, as shown in Figure 23. The APOXI instance accepts a MMI model sent by the Aragon tool, e.g. a screen layout, creates the corresponding window object with all child objects and sends back the resulting display image to Aragon. Aragon then merges the resulting image

with the invisible domain-model. Besides rendering the corresponding visual representation, some additional information, in particular positioning information of UI elements are extracted from the target platform and sent back to Aragon. This avoids reimplementation of rendering and layout algorithms, which are provided by the domain framework anyway.

The great advantage of this approach for MMI design is, that any divergence between the visual feedback gained in the design stage and the final appearance on the target device is eliminated. Aragon guarantees exact visual feedback according to the target device already in the modeling (or design) phase. Furthermore, the approach used by the Aragon tools is automatically extensible because new user interface elements available in a domain framework, e.g. APOXI, can be displayed in the Aragon editors without further modification of the tools. In addition, by replacing the small interface of Aragon to the target framework, Aragon can easily be adapted to any target platform.

## Code Generation

According to the DSM architecture, code generators transform models conforming to a DSL into target code or an intermediate representation which then is interpreted on the target. Aragon supports both forms of code generation, i.e., it allows transforming a window layout alternatively to resource files or to C++ source code. The former one is used for easier customization because resource files may be stored on flash memory of a mobile device and easily replaced. The latter one is more compact and can be loaded fast into memory, which is required for low-end mobile devices due to limited memory and CPU resources.

Because of the textual output of both forms, the Aragon code generators follow the transformation technique model-to-text [LJJ07]. For this technique two main approaches are available [CH03]: visitor-based and template-based. However, both approaches hinder extensibility by DSM users as required for Aragon. The reason is that template languages are often complex and visitors directly operate on the internal representation of a meta-model, which usually shall be hidden to DSM users.

As consequence, we have combined both techniques to a two-phase code generator, which can be extended by DSM users more easily:

1. Domain models given in XML are transformed by means of XSLT into an intermediate model (model-to-model transformation). The XSLT rules can be extended by DSM users.
2. A non-extensible visitor transforms the intermediate model into resulting resource files or C++ source code (model-to-text transformation).

Besides this, Aragon also provides template-based, non-extensible code generators that generate and update C++ source code for applications based on available meta-data of applications. This allows automatic synchronization of meta-model and C++ source code and increases productivity. In this way, common application code concerning registration and feature invocation can be automatically generated from meta-models.

Altogether, the architectural features as described above result in a flexible and extensible tool which, in its core, is independent from the target framework (which is currently APOXI). Actually, being almost APOXI agnostic and only having a tiny interface, Aragon is readily prepared to be used together with other frameworks for MMI design.

## 5.3 Textual Domain-Specific Languages

In this subsection, we present challenges and results from an application-oriented research project Testbed [HHK+08] aiming the development of a software testbed for unit/integration/system testing of mobile software frameworks.

The field of textual domain-specific languages is well elaborated, mainly from experience over five decades on textual general-purpose programming languages. This includes language definition (concrete and abstract syntax) as well as tool support.

Usually, the concrete syntax (CS) is defined in form of a context-free grammar and the abstract syntax (AS) is either explicitly defined and mapped to concrete syntax, implicitly derived from concrete syntax or concrete and abstract syntax are defined in single source [PP08]. On contrary, other approaches [JBK06, MFF+06] allows the description of a textual concrete syntax for a given abstract syntax in form of a meta-model.

Tool support for textual languages includes text-to-model transformation (parsing), editor support, and code generation. The automatically generation of compiler frontends including scanner and parser for text-to-model transformation is an established area and a lot of such tools (e.g. CoCo/R, LPG, ANTLR) are available. As a consequent further development, actual approaches as enumerated in [PP08] also automatically generate editor support from a context-free grammar definition and provide support for template-based or visitor-based code generation.

For the Testbed project, we followed a DSM approach by providing a textual domain-specific language to specify test cases that can be executed by a test environment. Domain-specific language for testing has been applied by various approaches (e.g. Sinha [SSM03] and Siddhartha [RR99]). In particular, the usage of DSLs for testing of mobile software frameworks has several advantages [HHK+08]:

- Testers usually are not familiar with source code of the system under test (SUT) even are not usually C++ experts.
- Languages like C++ cause many programming errors, most notable errors concerning memory management. Using a high-level testing language prevents many programming errors and facilitates more robust test cases that cannot harm the SUT.
- The use of a separate language (and not the language used to program the SUT) leads to decoupling of the SUT and test cases. Instead of using the API of a SUT directly, high-level language constructs defined by a DSL are more stable with regard of changes of the SUT.
- A DSL also facilitates high-level constructs for testing as well as of the problem domain.

We defined a textual domain-specific language that includes first-class concepts on language level for testing of mobile software frameworks. Besides general-purpose elements for assignment, loops, conditional statements, etc., the language provides following domain-specific elements:

- Statements to verify test results.
- Statements to control test case execution.
- Statements for logging.
- Statements to simulate the protocol stack for both sending and receiving signals of different communication protocols from a test case.

Figure 24 gives an (simplified) example of a test case intended to test a single function of the system under test. The instruction in line 2 simulates a function call to the SUT which in turn sends a signal (MN_ATTACH_REQ) to the protocol stack. This event is consumed and verified from the statement in line 3. To simulate the response back from the protocol stack to the SUT, an SDL signal is created and initialized (lines 4–6) and sent to the SUT in line 7. Triggered by this event, the SUT will call a callback function. This call is consumed and verified by the script in line 8. Test scripts written in the defined language are compiled into an intermediate code that is interpreted by the test environment which can be considered as domain framework in context of a DSM solution.

## End-User Programming 5.4

In this subsection, we present results from an application-oriented research project carried out by SCCH and the COMET partner company KEBA AG. The goal of this project was to develop a tool prototype that supports end users (e.g. machine operators) without detail software development expertise to program control programs of injection molding machines. Furthermore,

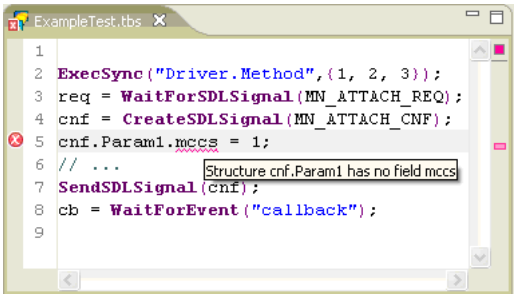



FIGURE 24 Notation and editing support for testing language.

the modification of control programs must be possible on the touch screen on the machine directly.

In general, end-user programmers (EUP) are people who write programs, but not as their primary job function [MKB06]. Instead, they must write programs in support of achieving their main goal, which is something else. End-users are often experts of a certain domain, like electrical engineering, robotics, or plastics. Such domain experts have to transfer their domain knowledge into a representation that can be understood by the computer.

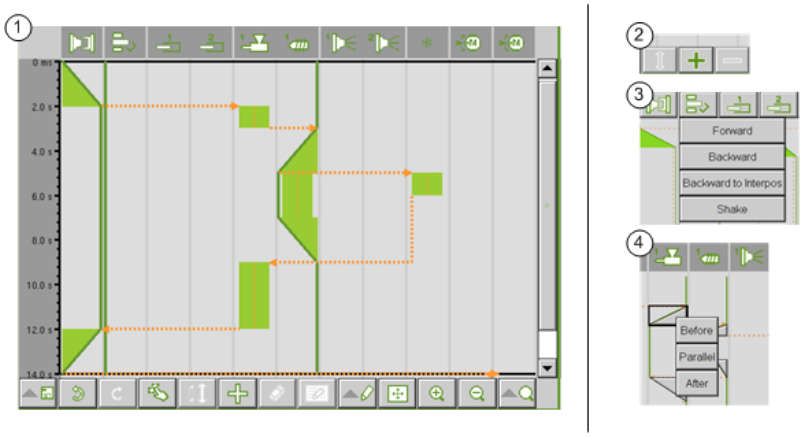
We have chosen a DSM approach for several reasons. As reported in [PHS+08a], current modeling notations and languages in the automation domain do not satisfy the requirements of a programming language, which can be used by domain experts. On contrary, domain-specific languages are a proven approach to bring programming closer to application domains [PHS+08a], and hence, to domain experts. By following a DSM approach, the challenge is to provide a language or notation that can be used by end-users on the machine directly. In this context, end-users are domain experts as machine operators and commissioning specialists.

## Language for End-User Programming

A domain-specific language for programming machine cycles of injection molding machines has to incorporate machine aggregates, aggregate actions, and blocks of sequential or parallel actions. On the meta-model level, individual aggregate actions of a machine connected together according to sequential and parallel executions result in the abstract syntax graph for an entire machine cycle.

To visualize a machine cycle to end users, the graph is not displayed directly but by means of a notation (concrete syntax) that arranges individual

nodes and their connections in a two dimensional way, as shown in Figure 25. Nodes representing aggregate actions are placed in horizontal columns resulting in a column for each aggregate. The chosen icons together with the order of columns corresponding to the aggregate position on the actual machine give more specific information for domain experts to identify individual actions of aggregates compared to general-purpose programming languages or software diagrams. Vertically, the actions are placed according to their de-




Notation, interaction, and tool support to manipulate machine cycles on touch screens. FIGURE 25

pendency starting with the very first action of a machine cycle on the top. The vertical dimension of a single action and, hence, of the entire graph, corresponds to the time required to execute an action, or the entire machine cycle respectively. This technique that maps the property duration of an action to a visual dimension facilitates to locate time-consuming actions and to identify actions along the critical path.

The described incorporation of domain aspects (e.g. aggregates and duration) as elements of a visual domain-specific language is a typical example how a DSL can facilitate end-user programming for domain exerts.

## Visualization and Interaction

As pointed out earlier, the success of a DSM solution largely depends on provided tool support to create and manipulate software models, e.g. machine cycles, in a given domain-specific language. From our experience, tool support

becomes much more important when interaction is done by end-users on touch screens compared to interaction on personal computers with keyboard and mouse pointer devices.

For instance, the modification of a machine cycle by inserting a new action is a non-trivial action on a touch screen that requires special guidelines on individual steps as shown in Figure 25. After pressing the insert button (2), the user has to perform several steps to insert a new action, whereas the editor gives visual feedback about the next step. First, the user selects the corresponding aggregate from the column headers and, afterwards, selects from available actions provided by the selected aggregate (3). Second, the user selects an already existing action and the relative insertion option (4).

For end-user programming, it is also important to achieve a fault tolerance for user interaction so that users can cancel operations at any time. The specification of end-users needs in form of personas [MK08] and usability evaluations provides valuable feedback for the design of both the notation as well as the interaction concepts.

## Acknowledgements

Research and development described in this chapter has been carried out by Software Competence Center Hagenberg GmbH (SCCH) in close cooperation with its scientific partners and its partner companies within the frame of the Austrian K*plus* and COMET competence center programs.

# References

[ABD+04] Alain Abran, Pierre Bourque, Robert Dupuis, James W. Moore, and Leonard L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2004 version edition, 2004.

[AGM05] Paris Avgeriou, Nicolas Guelfi, and Nenad Medvidovic. Software architecture description and uml. pages 23–32. 2005.

[AMS08] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage dsl embedded in haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 225–228, New York, NY, USA, 2008. ACM.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2 edition, November 2004.

[BAB+05] Stefan Biffl, Aybüke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher. *Value-Based Software Engineering*. Springer Verlag, oct 2005.

[Bac97] James Bach. Good enough quality: Beyond the buzzword. *Computer*, 30(8):96–98, 1997.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[BD04] Pierre Bourque and Robert Dupuis, editors. *SWEBOK - Guide to the Software Engineering Body of Knowledge, 2004 Version*. IEEE Computer Society, 2004 version edition, 2004.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[Bei90] Boris Beizer. *Software Testing Techniques 2E*. International Thomson Computer Press, 2nd edition, June 1990.

[BG00] Kent Beck and Erich Gamma. *More Java Gems*, chapter Test-infected: programmers love writing tests, pages 357–376. Cambridge University Press, 2000.

[Boe76] B. W. Boehm. Software engineering. *Transactions on Computers*, C-25(12):1226–1241, 1976.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[BR08] Armin Beer and Rudolf Ramler. The role of experience in software testing practice. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 258–265, Parma, Italy, 2008. IEEE Computer Society.

[Bru01] H. Bruyninckx. Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528 vol.3, 2001.

[BW08] Georg Buchgeher and Rainer Weinreich. Integrated software architecture management and validation. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, pages 427–436, 2008.

[BWK05] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, St. Louis, MO, USA, 2005. ACM.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, September 2002.

[CEC00] Krzysztof Czarnecki, Ulrich Eisenecker, and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.

[Cha05] R. N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.

[Chr92] Gerhard Chroust. *Modelle der SoftwareEntwicklung*. Oldenbourg Verlag München Wien, 1992. in German.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan C. Wills. *Domain Specific Development with Visual Studio DSL Tools (Microsoft .Net Development)*. Addison-Wesley Longman, Amsterdam, May 2007.

[CKK02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, January 2002.

[Cle95] Paul Clements. Formal methods in describing architectures. In *Monterey Workshop on Formal Methods and Architecture*, September 1995.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA, 1996. IEEE Computer Society.

[CMM06] CMMI for development, version 1.2. Technical report CMU/SEI-2006-TR-008, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA 15213-3890, August 2006.

[CN02] P. Clements and L. N. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Professional Series: The SEI Series in Software Engineering. Addison Wesley, 2002.

[Cop04] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, 2004.

[Cza05] Krzysztof Czarnecki. Overview of generative software development. pages 326–341. 2005.

[DeM86] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[Dio93] R. Dion. Process improvement and the corporate balance sheet. *IEEE Software*, pages 28–35, July 1993.

[DN02] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *Software Engineering, IEEE Transactions on*, 28(7):638–653, 2002.

[Dor93] Alec Dorling. Software Process Improvement and Capability Determination. *Software Quality Journal*, 2(4):209–224, December 1993. also in: Information and Software Technology, vol. 35, no. 6/7, June 1993, p. 404.

[Dor09] Bernhard Dorninger. ProMoRTE: A process model runtime environment based on OSGi. 2009. accepted for publication at 7th IEEE International Conference on Industrial Informatics (INDIN 2009).

[Dou02] Bruce Powell Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[DW06] Patricia Derler and Rainer Weinreich. Models and tools for soa governance. In *International Conference on Trends in Enterprise Application Architecture*. Springer Lecture Notes on Computer Science (LNCS), December 2006.

[Eck05] Wayne W. Eckerson. *Performance Dashboards: Measuring, Monitoring, and Managing Your Business*. Wiley, October 2005.

[EJB06] Enterprise javabeans 3.0 specification, 2006.

[EK07] Andrew D. Eisenberg and Gregor Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM.

[FG99] Mark Fewster and Dorothy Graham. *Software Test Automation*. Addison-Wesley Professional, September 1999.

[FJ99] Mohamed Fayad and Ralph Johnson. *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. John Wiley & Sons, October 1999.

[FPC97] William A. Florac, Robert E. Park, and Anita D. Carleton. Practical software measurement: Measuring for process management and improvement. Guidebook CMU/SEI-97-HB-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, April 1997.

[Gar84] David A. Garvin. What does product quality really mean? *Sloan Management Review*, 26(1):25–45, Fall 1984.

[Gar00] David Garlan. Software architecture: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM Press.

[GBU08] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of concrete textual syntax mapping approaches. pages 169–184. 2008.

[GJ96] Pankaj K. Garg and Mehdi Jazayeri, editors. *ProcessCentered Software Engineering Environments*. IEEE Computer Society Press, 1996.

[Gre07] J. Grenning. Applying test driven development to embedded software. *Instrumentation & Measurement Magazine, IEEE*, 10(6):20–25, 2007.

[Ham04] Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, Inc., October 2004.

[HB06] LiGuo Huang and Barry Boehm. How much software quality investment is enough: A Value-Based approach. *IEEE Software*, 23(5):88–95, 2006.

[HCR+94] James Herbsleb, Anita Carleton, James Rozum, Jane Siegel, and David Zubrow. Benefits of CMM-based software process improvement: Initial results. Technical Report CMU/SEI-94-TR-013, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, August 1994.

[Hel07] Hello2morro. Sonarj. http://www.hello2morrow.de, 2007.

[HHK+08] Walter Hargassner, Thomas Hofer, Claus Klammer, Josef Pichler, and Gernot Reisinger. A script-based testbed for mobile software frameworks. In *Proceedings of the First International Conference on Software Testing, Verification and Validation*, pages 448–457. IEEE, April 2008.

[HNS99] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Professional, November 1999.

[Hof05] Christine Hofmeister. Architecting session report. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 209–210, Washington, DC, USA, 2005. IEEE Computer Society.

[HSSL02] B. Henderson-Sellers, F. Stallinger, and B. Lefever. Bridging the gap from process modelling to process assessment: the OOSPICE process specification for component-based software engineering. In *Proceedings of the 28th Euromicro Conference*, pages 324–331. IEEE Computer Society, 2002.

[HT06] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, July 2006.

[Hum89] W. Humphrey. *Managing the Software Process*. AddisonWesley Reading Mass., 1989.

[Hum95] W. Humphrey. *A Discipline for Software Engineering*. SEI Series in Software engineering. AddisonWesley, 1995.

[IEE90] IEEE Std 610.12-1990: IEEE standard glossary of software engineering terminology, 1990.

[IMP05] P. Inverardi, H. Muccini, and P. Pelliccione. Dually: Putting in synergy uml 2.0 and adls. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 251–252, Washington, DC, USA, 2005. IEEE Computer Society.

[Int08] International Organization for Standardization (ISO). Systems and software engineering - architectural description working draft 3, 2008.

[ISO95] ISO/IEC 12207:1995, Information technology - Software life cycle processes, 1995. Amd.1:2002; Amd.2:2004.

[ISO98] ISO/IEC TR 15504-7:1998(e), Information technology - Software process assessment - Part 7: Guide for use in process improvement, 1998.

[ISO01] ISO/IEC 9126-1:2001, Software engineering - Product quality - Part 1: Quality model, 2001.

[ISO03] ISO/IEC 15504:2003, Information Technology - Process Assessment, 2003.

[ISO05] ISO/IEC 25000:2005, Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE, 2005.

[ISO09] ISO/IEC PDTR 29110:2009, Software Engineering - Lifecycle Profiles for Very Small Enterprises (VSE), January 2009.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997.

[Joh99] Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1 edition, September 1999.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing, 2002.

[KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, September 2004.

[KFN99] Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. Wiley, 2 edition, April 1999.

[Kin01] Atte Kinnula. *Software Process Engineering Systems: Models and Industry Cases*. Oulu University Press, 2001. ISBN 951-42-6508-4.

[KL05] A.G. Koru and H. Liu. Building effective defect-prediction models in practice. *IEEE Software*, 22(6):23–29, 2005.

[KOS06] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *Software, IEEE*, 23(2):22–30, 2006.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 3rd edition, 2003. ISBN 0321197704, 9780321197702.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated definition of abstract and concrete syntax for textual languages. pages 286–300. 2007.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, April 2003.

[KWZ09] Thomas Kriechbaum, Rainer Weinreich, and Thomas Ziebermayr. Compatibility rules for the homogeneous evolution of enterprise services. In *International Symposium on Service Science (ISSS)*, pages 189–200. Logos Verlag Berlin, March 2009.

[LBB+05] U. Löwen, R. Bertsch, B. Böhm, S. Prummer, and T. Tetzner. Systematisierung des Engineerings von Industrieanlagen. *atp - Automatisierungstechnische Praxis, Oldenbourg Industrieverlag*, (4):54–61, 2005. in German.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, November 1993.

[LH09] Henrik Lochmann and Anders Hessellund. An integrated view on modeling with multiple domain-specific languages. In *Proceedings of the IASTED International Conference Software Engineering (SE 2009)*, pages 1–10. ACTA Press, February 2009.

[LJJ07] B. Langlois, C. E. Jitia, and E. Jouenne. Dsl classification. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[LKT04] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[Lon93] J. Lonchamp. A structured conceptual and terminological framework for software process engineering. In *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the*, pages 41–53, Feb 1993.

[LR07] Stefan Larndorfer and Rudolf Ramler. TestCockpit: business intelligence for test management. In *Work in Progress Session in conjunction with 33rd EUROMICRO Conf. on Software Engineering and Advanced Applications*, 2007.

[LRB09] Stefan Larndorfer, Rudolf Ramler, and Clemens Buchwiser. Experiences and results from establishing a software cockpit. In *upcoming*, 2009.

[LRFL07] Stefan Larndorfer, Rudolf Ramler, Christian Federspiel, and Klaus Lehner. Testing High-Reliability software for continuous casting steel plants - experiences and lessons learned from siemens VAI. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 255–262, Luebeck, Germany, 2007. IEEE Computer Society.

[McF96] Bob McFeeley. IDEAL: A user's guide for software process improvement. Handbook CMU/SEI-96-HB-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, February 1996.

[MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Verlag, March 2008.

[MDT07] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, January 2007.

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. *Model-Driven Analysis and Synthesis of Concrete Syntax*. 2006.

[MGF07] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[MH04] Jürgen Münch and Jens Heidrich. Software project control centers: concepts and approaches. *Journal of Systems and Software*, 70(1-2):3–19, February 2004.

[Mil02] Dave Miller. *Fundamental Concepts for the Software Quality Engineer*, chapter Choice and Application of a Software Quality Model, pages 17–24. ASQ Quality Press, 2002.

[MK08] Jennifer Mcginn and Nalini Kotamraju. Data-driven persona development. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1521–1524, New York, NY, USA, 2008. ACM.

[MKB06] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 75–80, New York, NY, USA, 2006. ACM.

[MKMG97] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *Software, IEEE*, 14(1):43–52, 1997.

[MMYA01] H. Mili, A. Mili, S. Yacoub, and E. Addy. *Reuse-Based Software Engineering: Techniques, Organizations, and Controls*. Wiley-Interscience, 2001.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461, Shanghai, China, 2006. ACM.

[Obj07] Object Management Group. Uml superstructure specification v2.1.1. OMG Document Number formal/07-02-05 http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf, 2007.

[Obj08] Object Management Group. Software & systems process engineering metamodel specification, version 2.0. http://www.omg.org/spec/SPEM/2.0/PDF, April 2008.

[Ope08] OpenUP - Open Unified Process, 2008. http://epf.eclipse.org/wikis/openup/.

[OSG07] Osgi service platform release 4, 2007.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-02, Software Engineering Institute, Carnegie Mellon University, February 1993.

[PGP08] F. Pino, F. Garcia, and M. Piattini. Software process improvement in small and medium software enterprises: A systematic review. *Software Quality Journal*, 16(2):1573–1367, June 2008.

[PHS+08a] Herbert Prähofer, Dominik Hurnaus, Roland Schatz, Christian Wirth, and Hanspeter Mössenböck. Monaco: A dsl approach for programming automation systems. In *SE 2008 - Software-Engineering-Konferenz 2008*, pages 242–256, Munic, Germay, February 2008.

[PHS+08b] Herbert Prähofer, Dominik Hurnaus, Roland Schatz, Christian Wirth, and Hanspeter Mössenböck. Software support for building end-user programming environments in the automation domain. In *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*, pages 76–80, New York, NY, USA, 2008. ACM.

[PP04] Gustav Pomberger and Wolfgang Pree. *Software Engineering*. Hanser Fachbuchverlag, October 2004.

[PP08] Michael Pfeiffer and Josef Pichler. A comparison of tool support for textual domain-specific languages. *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7, October 2008.

[PP09] Michael Pfeiffer and Josef Pichler. A DSM approach for End-User Programming in the Automation Domain. 2009. accepted for publication at 7th IEEE International Conference on Industrial Informatics (INDIN 2009).

[PPRL07] Josef Pichler, Herbert Praehofer, Gernot Reisinger, and Gerhard Leonhartsberger. Aragon: an industrial strength eclipse tool for MMI design for mobile systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 156–163, Innsbruck, Austria, 2007. ACTA Press.

[PR08] Josef Pichler and Rudolf Ramler. How to test the intangible properties of graphical user interfaces? In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST 08*, pages 494–497. IEEE Computer Society, 2008.

[PRS00] G. Pomberger, M. Rezagholi, and C. Stobbe. *Handbuch für Evaluation und Evaluierungsforschung in der Wirtschaftsinformatik*, chapter Evaluation und Verbesserung wiederverwendungsorientierter Software-Entwicklung. Oldenbourg Verlag, München/Wien, 2000. in German.

[PRZ09] Guenter Pirklbauer, Rudolf Ramler, and Rene Zeilinger. An integration-oriented model for application lifecycle management. 2009. accepted for ICEIS 2009, 11th International Conference on Enterprise Information Systems.

[PSN08] R. Plösch, F. Stallinger, and R. Neumann. SISB - systematic improvement of the solution business: Engineering strategies for the industrial solutions business, version 1.0. Technical report, Software Competence Center Hagengerg, August 2008. (non-public project deliverable).

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, October 1992.

[Ram04] Rudolf Ramler. Decision support for test management in iterative and evolutionary development. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 406–409, Linz, Austria, 2004. IEEE Computer Society.

[Ram08] Rudolf Ramler. The impact of product development on the lifecycle of defects. In *Proceedings of the DEFECTS 2008 Workshop on Defects in Large Software Systems*, pages 21–25, Seattle, Washington, 2008. ACM.

[RBG05] Rudolf Ramler, Stefan Biffl, and Paul Grünbacher. *Value-Based Software Engineering*, chapter Value-Based Management of Software Testing, pages 225–244. Springer Verlag, 2005.

[RCS03] Rudolf Ramler, Gerald Czech, and Dietmar Schlosser. Unit testing beyond a bar in green and red. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003*, pages 10–12, Genova, Italy, 2003. LNCS.

[Roy70] W. W. Royce. Managing the development of large software systems:: Concepts and techniques. In *Proc. IEEE WESCON*, pages 1–9. IEEE, August 1970.

[RR99] Arthur A. Reyes and Debra J. Richardson. Siddhartha: a method for developing domain-specific test driver generators. In *In Proc. 14th Int. Conf. on Automated Software Engineering*, pages 12–15, 1999.

[RvW07] Ita Richardson and Christiane Gresse von Wangenheim. Why are small software organizations different? *IEEE Software*, 24(1):18–22, January/February 2007.

[RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[RW06] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91, Shanghai, China, 2006. ACM.

[RW08] Rudolf Ramler and Klaus Wolfmaier. Issues and effort in integrating data from heterogeneous software repositories and corporate databases. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 330–332, Kaiserslautern, Germany, 2008. ACM.

[RWS+09] Rudolf Ramler, Klaus Wolfmaier, Erwin Stauder, Felix Kossak, and Thomas Natschläger. Key questions in building defect prediction models in practice. In *10th International Conference on Product Focused Software Development and Process Improvement, PROFES 2009*, Oulu, Finnland, 2009.

[RWW+02] Rudolf Ramler, Edgar Weippl, Mario Winterer, Wieland Schwinger, and Josef Altmann. A quality-driven approach to web testing. In *Ibero-american Conference on Web Engineering, ICWE 2002*, pages 81–95, Argentina, 2002.

[Sad08] Daniel A. Sadilek. Prototyping domain-specific language semantics. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 895–896, New York, NY, USA, 2008. ACM.

[Sam01] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 2001.

[SB03] Douglas C. Schmidt and Frank Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, 2nd revised edition (rev). edition, January 2009.

[SCA07] Service component architecture specifications, 2007.

[Sch06] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[SDR+02] F. Stallinger, A. Dorling, T. Rout, B. Henderson-Sellers, and B. Lefever. Software process improvement for component-based software engineering: an introduction to the OOSPICE project. In *Proceedings of the 28th Euromicro Conference*, pages 318–323. IEEE Computer Society, 2002.

[Sha90] M. Shaw. Prospects for an engineering discipline of software. *Software, IEEE*, 7(6):15–24, Nov 1990.

[She31] Walter A. Shewhart. *Economic control of quality of manufactured product*. D. Van Nostrand Company, New York, 1931.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10):167–176, October 2005.

[Sof07] Software Tomography GmbH. Sotoarc. http://www.software-tomography.de/index.html, 2007.

[Som04] Ian Sommerville. *Software Engineering*. Addison Wesley, seventh edition, May 2004.

[SPP+06] F. Stallinger, R. Plösch, H. Prähofer, S. Prummer, and J. Vollmar. A process reference model for reuse in industrial engineering: Enhancing the ISO/IEC 15504 framework to cope with organizational reuse maturity. In *Proc. SPICE 2006, Luxembourg, May 4-5, 2006*, pages 49–56, May 2006.

[SPPV09] Fritz Stallinger, Reinhold Plösch, Gustav Pomberger, and Jan Vollmar. Bridging the gap between ISO/IEC 15504 conformant process assessment and organizational reuse enhancement. 2009. (accepted for SPICE Conference 2009, Software Process Improvement and Capability Determination, 2-4 June 2009, Turku, Finland).

[Spr08a] Spring dynamic modules for osgi(tm) service platforms, 2008.

[Spr08b] The spring framework - reference documentation, 2008.

[SPV07] F. Stallinger, R. Plösch, and J. Vollmar. A process assessment based approach for improving organizational reuse maturity in multidisciplinary industrial engineering contexts. In *Proceedings of ESEPG 2007, Amsterdam, 14th June 2007*, June 2007.

[SRA06] Christoph Steindl, Rudolf Ramler, and Josef Altmann. *Web Engineering: The Discipline of Systematic Development of Web Applications*, chapter Testing Web Applications, pages 133–153. Wiley, 2006.

[SSM03] A. Sinha, C. S. Smidts, and A. Moran. Enhanced testing of domain specific applications by automatic extraction of axioms from functional specifications. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 181–190, 2003.

[Ste00] David B. Stewart. Designing software components for real-time applications. In *in Proceedings of Embedded System Conference*, page 428, 2000.

[Tas02] Gregory Tassy. The economic impacts of inadequate infrastructure for software testing, NIST planning report 02-3, May 2002.

[Tia05] Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley & Sons, 1., auflage edition, February 2005.

[TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. pages 198–209. 2005.

[TMD09] R. N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[TvdH07] Richard N. Taylor and Andre van der Hoek. Software design and architecture the once and future focus of software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 226–243, Washington, DC, USA, 2007. IEEE Computer Society.

[V-M06] V-Modell XT, part1: Fundamentals of the V-Modell XT, version 1.2.1. Technical report, 2006. http://www.v-modell-xt.de/.

[vDKV00] Arie v. van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, March 2002.

[Voa08] Jeffrey Voas. Software quality unpeeled. *STSC CrossTalk*, (Jun 2008):27–30, 2008.

[VRM03] M. Venzin, C. Rasner, and V. Mahnke. *Der Strategieprozess - Praxishandbuch zur Umsetzung im Unternehmen*. 2003. in German.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development : Technology, Engineering, Management*. John Wiley & Sons, June 2006.

[Was96] A.I. Wasserman. Toward a discipline of software engineering. *Software, IEEE*, 13(6):23–31, Nov 1996.

[Was06] Hironori Washizaki. *Product-Focused Software Process Improvement*, volume 4034 of *Lecture Notes in Computer Science*, chapter Building Software Process Line Architectures from Bottom Up, pages 415–421. Springer Berlin / Heidelberg, 2006.

[WH05] Eoin Woods and Rich Hilliard. Architecture description languages in practice session report. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 243–246, Washington, DC, USA, 2005. IEEE Computer Society.

[WV02] James A. Whittaker and Jeffrey M. Voas. 50 years of software: Key principles for quality. *IT Professional*, 4(6):28–35, 2002.

[WWZ07] Rainer Weinreich, Andeas Wiesauer, and Thomas Ziebermayr. A component model for integrating remote applications and services via web portals. *Journal of Object Technology (JOT)*, 6(8), September 2007.

[WZ05] Rainer Weinreich and Thomas Ziebermayr. Enhancing presentation level integration of remote applications and services in web portals. In *2005 IEEE International Conference on Services Computing (SCC'05)*, volume 2, pages 224–236, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[WZD07] Rainer Weinreich, Thomas Ziebermayr, and Dirk Draheim. A versioning model for enterprise services. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, volume 2, pages 570–575, Los Alamitos, CA, USA, 2007. IEEE Computer Society.