Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,2
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Software Evolution for Industrial Automation Systems: Literature Overview

Johannes Stammel, Zoya Durdik, Klaus Krogmann
Roland Weiss, Heiko Koziolek

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

ABB DARWIN

Software Evolution for Industrial Automation Systems: Literature Overview

Johannes Stammel, Zoya Durdik, Klaus Krogmann
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe

Roland Weiss, Heiko Koziolek
Industrial Software Systems, ABB Corporate Research
Wallstadter Str. 59, 68526 Ladenburg

Fakultät für Informatik, Karlsruher Institut für Technologie (KIT), Interner Bericht 2011-2

February 7, 2011

Outline

In this document we collect and classify literature with respect to software evolution. The main objective is to get an overview of approaches for the evolution of sustainable software systems with focus on the domain of industrial process control systems.

The document is structured as follows: In Chapter 1 we describe the literature investigation strategy. We present questions with respect to the problem space (software evolution) and the solution space (solution approaches), and a set of evaluation criteria. Chapter 2 surveys the problem domain and explains our understanding of sustainability and evolution. In Chapter 3 we summarise analytical approaches, which aim at identifying evolution problems or factors that influence evolution. In Chapter 4 we describe solution approaches for active prevention or handling of evolution challenges. Chapter 5 gives a quick overview on related survey documents. Chapter 6 briefly summarises the findings from this document. Chapter 7 lists information sources which were used for this document and Chapter 8 lists the definitions of important terms used in this document in a glossary.

Contents

Chapter 1

Motivation and Objectives

1.1 Objectives

This document contains a literature overview for the DARWIN research project. It is a living document and is regularly reviewed and extended. The ABB project manager, Roland Weiss, and FZI Project Manager, Klaus Krogmann, are responsible for upkeep of this document.

The goal of this document is to provide an overview of various strategies concerning evolution of sustainable systems. The selection criteria for the survey include the applicability of the proposed approaches to real-world evolution scenarios and systems. The applicability is determined based on the availability of industrial experience reports and tooling.

The research areas are derived from questions covering the problem domain of evolution and migration in general, with special attention to sustainable systems. The solution space includes analytical solutions, proactive solutions (prevention and avoidance of problems), and reactive solutions (migration, refactoring, reengineering, and evolution).

1.2 Investigation Process

Our investigation activities in order to come up with this document comprise the following high-level investigation steps:

1. Understand and characterise evolution problems
2. Derive a classification scheme for problems and solution strategies
3. Collect solution strategies and fill in the classification scheme

The following selection criteria are applied for the surveyed approaches:

1. Books are used as a starting point as they carry fundamental and well-established knowledge.
2. Dissertations serve for the knowledge transfer, as they provide an innovative and well-detailed approach to a problem.
3. Journal articles provide an overview on more recently developed approaches which are usually mature, since a rigorous review process is passed and the demands for validations are high.
4. Articles from conferences and workshops provide novel approaches to the problem, however there might be fewer expert review, validation, or practical experience.

For an extended list of reviewed information sources (conferences, articles, books) and search keys refer to Section 7.

This documents contains chapters and sections with a survey character and those which discuss a single approach in more detail. In overview chapters, the literature and references are sorted according to the two criteria: suitability to provide an *overview* and the *relevance* of literature and references. Survey articles with high relevance are listed first, literature on single approaches with less relevance in the end of the listing.

1.3 Investigation Questions

Our literature investigation is guided by these two sets of questions:

- Questions regarding the understanding and description of evolution problems;
- Questions regarding the description and evaluation of solution strategies.

The corresponding questions sets are provided in the subsections 1.3.1 and 1.3.2.

1.3.1 Understanding and description of evolution problems

What are the characteristics of sustainable systems? Sustainable systems have several problems and peculiarities: Such systems are existing and running for at least 10 years and some of them could reach more than 30 years. During the lifetime, various changes need to be done to the system.

These changes are triggered by:

1. Changes of hardware devices (e.g. as old devices get outdated)
2. Changes of communication standards (new state-of-the-art)
3. Changes of user behaviour (e.g. more/other devices, increased usage behaviour, usability issues)
4. New customer requirements
5. New laws and regulations
6. Deprecation of surrounding software (e.g., OS, compilers, middleware, databases)
7. New development approaches (e.g., agile methods, model-driven techniques)

Dependencies to external technologies in these systems build a high technology stack (e.g. Windows XP, Java 1.5, Tomcat 5.5). During the long lifetime, a high team fluctuation, bad knowledge transfer, and distributed development lead to a loss of knowledge about the system (this also holds for other domains, but it is a factor which has to be considered).

Therefore, there the following questions arise:

- What are typical migration and evolution difficulties and problems?

  Typical migration and evolution difficulties and problems are reviewed in Section 2.

- How can these difficulties and problems be identified?

  Approaches for identifying such problems are presented in Section 3.

- Which approaches can be used to prevent and/or to solve these difficulties and problems (preventive and reactive)?

  Approaches for preventing and/or dealing with migration and evolution difficulties and problems are listed in Section 4.

1.3.2 Description and evaluation of solution strategies

Questions regarding the description and evaluation of solution approaches are the following:

- In which domains are these approaches applied?
- What are the benefits? What are the problems?
- What is the maturity of these approaches (existing tools, case studies, etc.)?

These questions are discussed together with the evaluation criteria, which are explained in detail in Section 1.4.

1.4 Evaluation Criteria

The identified approaches are evaluated with respect to a set of evaluation criteria.

The criteria catalogue covers properties like applicability, relevance, positive/negative perspective, degree of formalization, abstraction level, and development phase. A detailed description of these criteria is provided in the Table 1.1.

Criteria	Description
Development Phase:	Determines whether an approach is applicable for design, implementation, or maintenance. (Design / Implementation / Maintenance)
Relevance, Automation:	Specifies relevance for the automation domain. • Approaches specific to other domains (than automation) are marked as low relevant. • Approaches which are moderate beneficial or have some restrictions are marked as medium relevant. • Approaches bringing special benefits or suiting special conditions of the automation domain are marked as highly relevant. (Low / Medium / High)
Relevance, Sustainability:	Specifies the relevance from the sustainability perspective. Sustainability is the ability for a (software) system to be able stay alive for a long time (more than 10 years) and requires the ability to cope with a changing environment and changing user requirements during the whole period of life. These criteria indicate if an approach may have a high, medium or low impact on sustainability properties of a system. For example, an approach aiming to prevent architecture erosion or to empower knowledge transfer may be highly relevant. (Low / Medium / High)
Applicability:	Concerns the development stage of the approach regarding its applicability in industry. The applicability is determined by the availability of the technology, maturity of tools, and industrial experience with the approach. For example, if a tool suite is available which has already been used in industry, which is confirmed by industrial experience reports, this indicates a high applicability. • High applicability is given to approaches that are validated and supported by a tool. • Medium applicability is assigned if the approach is validated, but has no/weak tool support or that an application of approach/tools is connected to some risk. • Low applicability applies if only an idea of an approach exists and it still has to be validated and/or the application of it is could imply a high overhead. (Low / Medium / High)
Tool:	An URL to a supporting tool (if it exists). (URL)
Preventive / Reactive / Analytical:	Classifies if approaches are used to analyse problems to sustainability, their sources, prevent of problems, or whether they can be used to solve problems. (Preventive / Reactive / Analytical)

Formalization:	Concerns the form in which an approach is described, for example, for further automation. It is ranging from informal "best practices" to formal metrics or pattern systems. This criterion might not be applicable for some approaches. For example, software metrics are quite formalised. On the other hand best practices for software design are informal. (Formal / Informal / Not applicable)
Perspective:	Distinguishes if an approach actively increases the quality of the source code or software architecture with respect to sustainability (e.g. design patterns) or reduces negative influence factors (e.g. bad smell detection). This criterion might not be applicable for some approaches. (Positive / Negative / Not applicable)
Abstraction level:	Describes the level of abstraction at which the approach is applied; the following levels are considered: software development process (high level of abstraction), architectural and design (medium level of abstraction) or code (low level of abstraction). This criteria might not be applicable for some approaches. (Low / Medium / High / Not applicable)
Benefit for sustainable systems:	Briefly describes the benefit of the proposed approach with focus on the sustainability of systems.

Table 1.1: Evaluation criteria

Chapter 2

Background: Terms and Scope

The topic of evolution with respect to sustainable systems is accompanied with a very broad problem space. We first provide a break down of this problem space.
The following issues may have a negative impact on the sustainability of software systems [Som06]:

- Undefined (insufficiently defined) management process, project management in particular
- Time and budget pressure
- Outsourcing or off-shoring decisions
- Make or buy decisions
- Size and complexity of technology stack, dependency to multiple technologies
- Different evolution cycles of technology
- Unclear project roles and team organization
- Fluctuation in teams
- Understandability problems, lack of documentation, inconsistent or outdated artefacts
- Bad internal software quality, e.g., bad smells and antipatterns
- Weak quality assurance, insufficient testing
- Customer requirements to the system, e.g., no downtime allowed.

The following levels for addressing the above mentioned issues can be identified:

- Structural decisions
- Management and team organisation
- Development environment
- Automation strategies

In this document we investigate the state of the art for solutions regarding the mentioned sustainability issues. For each level we present a selection of approaches.

2.1 Software Evolution

There are multiple definitions of software evolution which are summarised in [TRDL07] (conference paper):

- The Research Institute on Software Evolution defines software evolution as: the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way [TRDL07].
- Lehman and Ramil define software evolution as: all programming activity that is intended to generate a new software version from an earlier operational version [LR00] (journal paper).
- Chapin defines software evolution as: the application of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version (...) together with the associated quality assurance activities and processes, and with the management of the activities and processes [CHK+99] (journal paper).

Therefore, under software evolution we will understand the following:

Software evolution is a change process of a software concerning both hardware and software starting from its development and going on until system retirement, during which the system changes into a different and usually more complex or better state. System evolution is part of the system life cycle.

Software evolution (the change process) may be caused through several reasons, such as:

- New requirements (change requests) to the system
- Evolution of the technology stack (especially in the case of strong coupling to it) causes the co-evolution of certain system parts
- Change of the environment

The ability to evolve a software rapidly and reliably preserving the architectural integrity of an application is a challenge for every organization. However, the evolution of a software system may become very cost and work expensive, this is usually caused by so-called "software erosion". *Software erosion* is the decreasing quality of the internal structure of a software system. It may occur already at early development stages of the system [BSB08]. If the system, despite of its degraded quality, is still valuable to its stakeholders, it is called *legacy system* [BSB08].

Software erosion may be for example caused by:

- Unmanaged or unstructured introduction of new features (processing of change requests or bugs)
- Unmanaged or unstructured changes of the system
- Unclear or outdated system architecture or bad system development, e.g., software redundancy through "copy and paste programming" or violation of architectural decisions
- Loss of knowledge about the system by team fluctuation, or insufficient or outdated documentation

The problem of software erosion is especially relevant for long living systems, as they are exposed to the change process during a very long period of time.

There are Workshops dedicated to the domain of sustainable systems: Workshop for sustainable and sustainable systems in the frame of SE Conference 2009 and the annual design for future workshop on sustainable software systems of the GI working group (L2S2).

As can be seen from the headings of the workshop contributions, a wide area of topics is covered for sustainable software: Model-Driven Solutions, Model-Based Solutions, Application Landscapes, Component Protocol Checking, Patterns and Pattern Systems, Modelling Evolving Systems, Variant Management and Co-Evolution of Requirements and Architecture, Learning from Open Source Development, Style-Based Architectures, Program Understanding, Application Landscapes, Models, Components, Patterns, Self-Adapting Systems, Embedded Models, Code Repositories and Model-Based Development, Architectural Maintainability Prediction, Distribution of Software Product Lines, Co-Evolution (architecture, embedded), Domain Specific Models, Life cycle Management, and Knowledge discovery and preservation.

References:

- [T. Mens. Software Evolution. Springer Berlin Heidelberg, 2008] [Men08]. (book)
- [C. Bommer, M. Spindler, and V. Barr. Software Wartung. dpunkt.verlag, 2008] [BSB08]. (book)
- [Jingwei Wu. Open Source Software Evolution and Its Dynamics. PhD thesis, The University of Waterloo, 2006] [Wu06]. (PhD thesis)
- [H. Malik, A.E. Hassan. Supporting software evolution using adaptive change propagation heuristics. IEEE International Conference on Software Maintenance, 2008. ICSM 2008., pages 177 – 186, 2008.] [MH08]. (conference paper)
- [S. Brcina, R. Bode, and M. Riebisch. Optimisation Process for Maintaining Evolvability during Software Evolution. 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009. ECBS 2009., pages 196 – 205, 2009] Discusses evolvability, and introduces a quality model for it. Presents a meta-model-based process for controlling and optimizing the evolvability characteristics of software baselines [BR09]. (conference paper)
- [M. Torchiano, F. Ricca, and A. De Lucia. Empirical Studies in Software Maintenance and Evolution. IEEE International Conference on Software Maintenance, 2007. ICSM 2007., pages 491 – 494, 2007] Working session on empirical studies in maintenance and evolution [TRDL07]. (conference paper)
- [E. Burd, S. Bradley, and J. Davey. Studying the Process of Software Change: an analysis of software evolution. Seventh Working Conference on Reverse Engineering, 2000. Proceedings., pages 232 –239, 2000] Describes the analysis and results of studies in software evolution. The main conclusions: 1) fewer software releases tend of lead to slower increases in data complexity 2) best people should be assigned to maintenance 3) preventative maintenance needs to be continuous theme [BBD00]. (conference paper)
- [P. Sneed, H.M. Brossler. Critical success factors in software maintenance: a case study. International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., pages 190 – 198, 2003] Tries to answer the questions: what is success in software maintenance and what factors have the greatest influence on the success of maintenance and evolution [SB03]. (conference paper)
- [U. Vora. Architectural Design Methodologies for Complex Evolving Systems. 12th IEEE International Conference on Engineering Complex Computer Systems, 2007., pages 197 – 206, 2007] Discusses the impact of evolution on the architectural design of a system designed using Aspect Oriented Design Methodology and a system designed using the Framework (CFFES) proposed by the authors [Vor07]. (conference paper)
- [M. M. Lehman, J. F. Ramil. Software evolution in the age of component-based software engineering. IEE Proceedings - Software, 147, pages 249 – 255, 2000] [LR00]. (journal paper)
- [N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, Wui-Gee Tan. Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 2, pages 3 – 30, 1999] [CHK+99]. (journal paper)

2.2 Software Migration

Software migration is a change process during which a software system is being moved from one environment or technology (meaning both, HW and SW) to another, [Men08] (book). This process is usually triggered either by a change request or by an own life cycle of environment or technology. Migration is a variant of *reengineering* in which the transformation is driven by a major technology change. Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form, [CI90] (journal paper). Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some kind of forward engineering or restructuring.

In the automation domain the term migration is usually associated with what we define as DCS migration.

DCS migration is the transfer process of a control system to a newer (or some other) version of the DCS, or to a competitor DCS. DCS migration is beyond the scope of this document. DCS migration includes application migration that is responsible for moving applications from one control platform to another one.

References:

- [T. Mens. Software Evolution. Springer Berlin Heidelberg, 2008] [Men08]. (book)
- [C. Bommer, M. Spindler, and V. Barr. Software Wartung. dpunkt.verlag, 2008] [BSB08]. (book)
- [J. Bisbal, D. Lawless, R. Richardson, D. O'Sullivan, B. Wu, J. Grimson, and V. Wade. A Survey of Research into Legacy System Migration. Technical report, Trinity College and Broadcom Eireann Research, Dublin, Ireland, 1997] [BLR+97]. (technical report)
- [L. Wu, H. Sahraoui, and P. Valtchev. Coping with legacy system migration complexity. 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings., pages 600 – 609, 2005] Coping with Legacy System Migration Complexity [WSV05]. (conference paper)
- [Ying Zou. Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms. PhD thesis, The University of Waterloo, 2003] [Zou03]. (PhD thesis)
- [E. J. Chikofsky , J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7, page 13, 1990] [CI90]. (journal paper)

2.3 Properties of the Automation Domain

The *automation domain* consists of automated plants, factories and utilities and their control systems. Such control systems usually contain real-time components and have client-server or multi-tier architectures with event-driven communication. They consist of distributed server nodes, client nodes, and embedded systems, e.g., controllers and field devices.

The automation domain has several properties:

- Strict requirements on the systems' availability: downtimes have to be minimised and ideally aligned with downtimes of machines or plants. Therefore software updates are often combined with hardware updates.
- Systems have to be developed following several industrial standards, which contribute additional requirements and constraints on the architecture and design. The relevant standards are briefly described in Section 2.5.
- Automation systems are usually developed based on the existing versions of the system (previous generations). Therefore, there is a danger of inheritance of existing problems regarding sustainability in the new systems.
- Automation systems have to handle a large number of embedded devices of different types. Therefore, there are dependencies to the technology stack and various devices and their evolution due to changing communication and integration standards.
- The same is true for the software systems as they are often build relying on 3rd party components (DB, HMI, Virtualization, component technology), which reflects in the technology stack.
- The increasing number and complexity of processes to be controlled and a therefore increasing complexity of the control systems has to be dealt with.
- A limited number of customers and thus less feedback compared to a mass software, as well as limited opportunities to learn from mistakes and to improve systems are typical.
- Automation systems are usually built on much more restricted budgets than military or aerospace systems.
- Changes of system's user profile, which demands new functionality, new technologies, and advanced look-and-feel are, due to the long-term use of these systems, are common.

Existing systems in the automation domain suffer from expensive evolution and maintenance as well as from long release cycles because sustainability was often not well considered during their construction.

2.4 Meaning of "Long living" and "Sustainability" in the Automation Domain

By *long living* software systems we understand systems having a life cycle of more than 10 years. The life cycle ranges from installation, engineering, and deployment to the final shut-down of the system.

Within their lifespan, these systems have to be adapted in order to react to environmental changes (standards, technologies) or user change requests (bugs, features), or to reflect changes of underlying hardware devices or communication standards. System users expect the system to provide support for more and diverse devices and to cope with increased plant sizes (more devices connected, more control loops executed, etc).

Among the common sustainability troubles in automation systems are dependencies to standard information technology, e.g. component systems (COM), .NET, or graphics subsystems. Their evolution cycles are usually shorter than the one of the process control system software.

Moreover, there is a high quality assurance effort involved in system changes. This is usually caused by laws, regulations and requirements on safety, changes of the development process and / or system environment, and the resulting very high testing effort.

By *sustainable* software systems we understand systems that can be maintained as product and that deployed cost effectively over total lifetime.

References:

- Working group "Sustainable Software Systems" (AK L2S2) (in German) http://akl2s2.ipd.kit.edu/.
- [F. J. R. Rojo, R. Roy, and E. Shehab. Obsolescence management for long-life contracts: state of the art and future trends. The International Journal of Advanced Manufacturing Technology, 2009] Provides a literature review on the problem of obsolescence in "sustainment dominated systems" in the aerospace domain. Clarifies and classifies issues regarding an obsolescence management planning (electronic components, mechanical components, software, materials, skills and tooling) [RRS09]. (conference paper)

2.5 Relevant Standards

There are plenty of predefined standards for use in the automation domain and therefore relevant for ABB. They influence the system architecture, which means that further structural decisions and modifications should be applied under consideration of these standards. In this section, we present several relevant standards: OPC UA, OPC, IEC 61508, and IEC 61850. This list should not be considered to be comprehensive. From the sustainability perspective, the list gives an impression on the kinds of dependencies that arise from standards and the kinds of APIs that are used in the automation domain (at ABB).

2.5.1 OPC

OPC is the abbreviation for an open connectivity standard in industrial automation and enterprise systems that support industry. Interoperability is assured through the creation and maintenance of open standard specifications. It is COM-based (COM – Component Object Model, Microsoft).

OPC specifications include:

- OPC Data Access
- OPC Alarms and Events
- OPC Batch
- OPC Data eXchange
- OPC Historical Data Access
- OPC Security
- OPC XML-DA
- OPC Complex Data
- OPC Commands
- OPC Unified Architecture

The effect on evolution resulting from this standard is the application of COM as a base technology and, therefore, a dependency to Microsoft's operating system families and technology stack.

References:

- OPC: http://www.opcfoundation.org

2.5.2 OPC UA

The **Unified Architecture (UA)** is the next generation OPC standard that provides a cohesive, secure and reliable cross platform framework for access to real time and historical data and events. One of the reasons for the establishment of this standard is that Microsoft has de-emphasized COM in favour of cross-platform capable Web Services and SOA (Service Oriented Architecture).

The Unified Architecture (OPC-UA) is described in a layered set of specifications broken into parts in order to isolate changes in OPC-UA from changes in the technology used to implement it.

The following parts comprise the OPC Unified Architecture specification:

- Concepts

- Security
- Address Space
- Services
- Information Model
- Mappings
- Profiles
- Data Access
- Alarms and Conditions
- Programs
- Historical Access
- Discovery
- Aggregates

The Unified Architecture is designed specifically to allow object and information models defined by others (vendors, end-users, other standards ...) to be exposed without alteration by OPC-UA servers.
The following Unified Architecture information model companion specifications have been developed:

- OPC UA For Devices (DI) Companion Specification;
- OPC UA For Analyser Devices (ADI) Companion Specification;
- OPC UA For IEC 61131-3 Companion Specification.

The effect on evolution resulting from this standard is the application of SOA and Web Services in order to enable platform independence. Compared to OPC, sustainability in terms of decoupling of layers is explicitly addressed.
References:

- OPC UA: www.opcfoundation.org/UA/

2.5.3 IEC 61850

IEC 61850 – the international standard for communication networks and systems in substations, defines the communication between devices in substations and related system requirements. It supports substation automation functions as well as their engineering.

Its goal is to support systems built from multi-vendors intelligent electronic devices (IEDs) networked together to perform protection, monitoring, automation, metering, and control. The standard enables the integration of the equipment and systems for controlling the electric power into complete system solutions and ensures interoperability of equipment by powering compatibility between interfaces, protocols, and data models. It is Ethernet based, provides device-to-device and station-to-station networks and operates as a client to control the network and talk to all the servers or slaves on the network.

IEC 61580 standard parts:

- IEC 61850-1 Introduction and overview

- IEC 61850-2 Glossary
- IEC 61850-3 General requirements
- IEC 61850-4 System and project management
- IEC 61850-5 Communication requirements for functions and devices models
- IEC 61850-6 Configuration description language for communication in electrical substations related to IEDs
- IEC 61850-7-1 Basic communication structure for substation and feeder equipment – Principles and models
- IEC 61850-7-2 Abstract communication service interface (ACSI)
- IEC 61850-7-3 Common data classes
- IEC 61850-7-4 Compatible logical node classes and data classes
- IEC 61850-7-410 Hydroelectric power plants - Communication for monitoring and control.
- IEC 61850-8-1 Specific communication service mapping (SCSM) - Mappings to MMS (ISO/IEC 9506-1 and ISO/IEC 9506-2) and to ISO/IEC 8802-3
- IEC 61850-9-1 Sampled values over serial unidirectional multi drop point to point link
- IEC 61850-9-2 Sampled values over ISO/IEC 8802-3
- IEC 61850-10 Conformance testing

The effect on evolution resulting from this standard is the enabled interoperability of multi-vendor IEDs, and therefore, lower costs to install, configure, and maintain such devices. However, the standard is based on Ethernet technology and therefore inherits its limitations, such as a complicated troubleshooting and change processes (the latter usually resulting in downtimes).
References:

- IEC 61580: http://www.abb.com/cawp/seitp202/c1256a8c00499292c1256d410038e215.aspx

2.5.4 IEC 61508

IEC 61508 – a standard for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems provides general guidance on the selection of design techniques according to the safety criticality of a software element under design.

IEC 61508 standard parts:

- Part 1: General requirements (required for compliance)
- Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems (required for compliance)
- Part 3: Software requirements (required for compliance)
- Part 4: Definitions and abbreviations (supporting information)
- Part 5: Examples of methods for the determination of safety integrity levels (supporting information)
- Part 6: Guidelines on the application of parts 2 and 3 (supporting information)
- Part 7: Overview of techniques and measures (supporting information)

The effect on evolution resulting from this standard is the enhanced safety and therefore reduced costs for the maintenance (decreased fault tolerance of devices if no danger exists in the process interface). IEC 61508 designs require maintenance throughout the product lifecycle and thus manufacturers are responsible for the documentation and certifications. On the other hand, IEC 61508 requires a strict development processes and costly recertification, which hinders evolution.

References:

- IEC 61508: http://www.iec.ch/zone/fsafety/

Chapter 3

Identifying and Analysing Evolution Problems

Outline

The approaches for the understanding and analysis of the evolution problems can be divided into the four major groups (no claim for completeness) (see Figure 3.1).

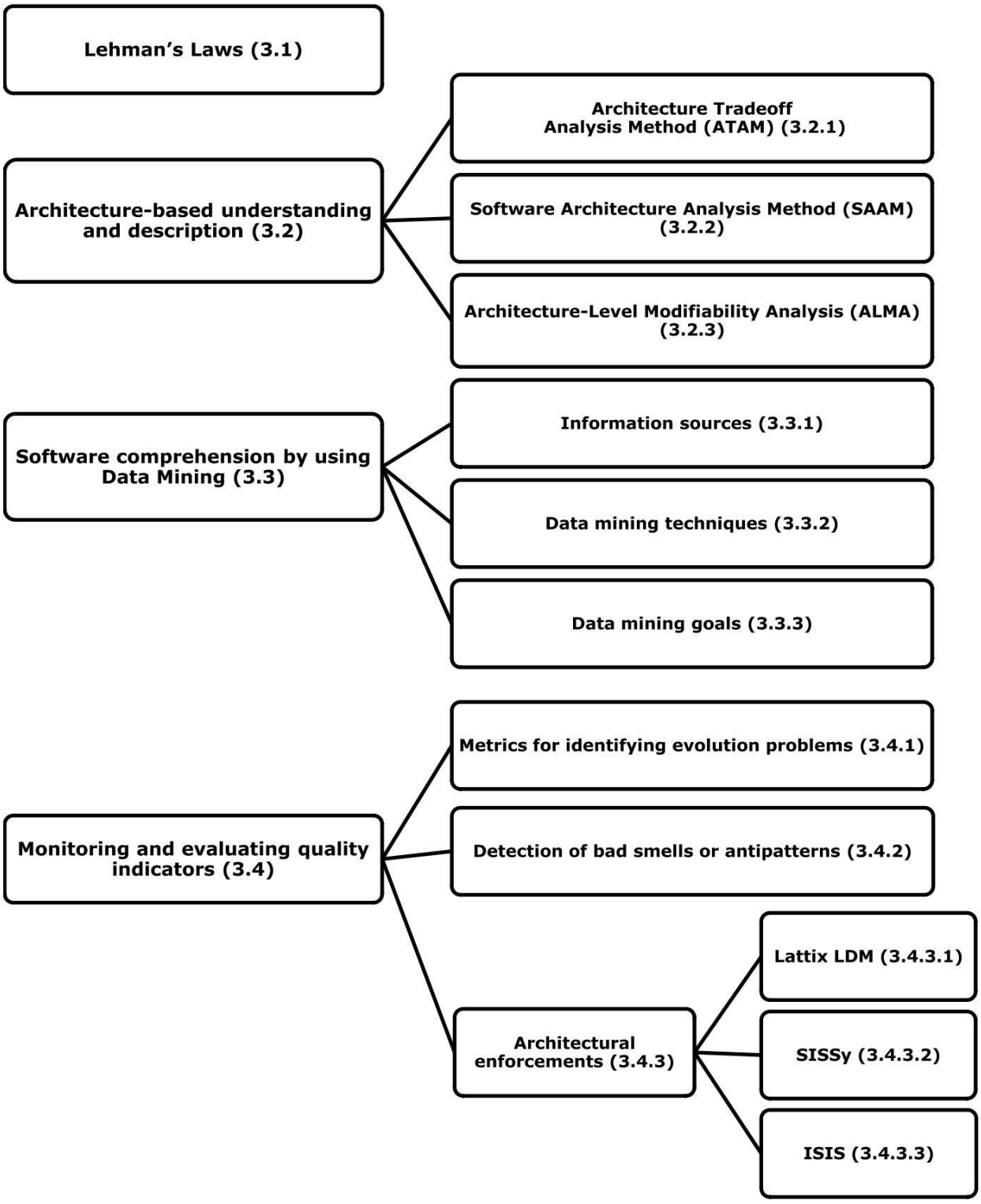

Figure 3.1: Classification of approaches for analysis of evolution problems (Selection of approaches presented in this overview)

Lehman's Laws are general laws of evolution, which provide a fundamental understanding of the evolution process of software systems. Although not all the laws are valid for different software systems types, they are still useful to be considered when developing systems aimed to be sustainable. An overview of Lehman's Laws is provided in Section 3.1.

Architecture-based understanding and description provides a set of analysis approaches, which can be applied to analyse an impact of planned evolution steps. These approaches are executed at the architectural level. There is a plenty of methods, however, only the following the most famous methods are described:

- Architecture Tradeoff Analysis Method, ATAM, Section 3.2.1
- Scenario-Based Architecture Analysis Method, SAAM, Section 3.2.2
- Architecture-Level Modifiability Analysis, ALMA, Section 3.2.3

Software comprehension by using historical data mining uses various information repositories (e.g. code repositories, mailing lists) in order to discover hidden dependencies in the data (e.g. change propagation) or to predict possible future development scenarios of the data (e.g. bug prediction). Among others, understanding (large) software systems, propagating changes, and predicting and identifying bugs are the most interesting research directions, when concerning sustainable systems. Here, change propagation, and bug prediction and identification could support maintenance. Understanding of the system could compensate lack of documentation. Section 3.3 provides an introduction into the topic and relevant references. Besides the historical data mining, that is described here, there are other types of mining data, e.g., web mining, text mining, and mining of statistical data that are less relevant for sustainable software systems.

Quality indicators indicate state of the implementation or design of a system and may help to identify possible problems. Regular monitoring of quality indicators and correction of identified problems helps to minimize overall maintenance overhead of sustainable systems. Beside that, it improves the overall system quality.

There are two types of indicators: quantitative (numeric representations and have to be interpreted in order to derive the semantic rationale of the problem) and qualitative (provide semantic rationale about the problem itself) indicators.

Subsection 3.4.1 explains and provides a brief overview of product, process and project metrics. Metrics are helpful indicators for the state of the system and may be used in order to indicate possible troubles. Subsection 3.4.3 introduces dependency analysis tool (Lattix LDM) and intern quality analysis tools (SISSY, ISIS). Such tools provide partial automated support for monitoring system state (e.g. based on metrics). Subsection 3.4.2 describes the concepts of bad smells and antipatterns. These are indicators for possible troubles in the system on architectural and code levels.

Most of the quality indicators should be interpreted carefully. What is a sign for a bad design in one case (antipattern), might be actually a pattern solving some specific problem in a given context (e.g. Façade pattern).

All practicable approaches (beside the overview sub/sections and approaches from low relevant sections) are evaluated according to the evaluation criteria. The evaluation result is summarised in Figure 3.2.

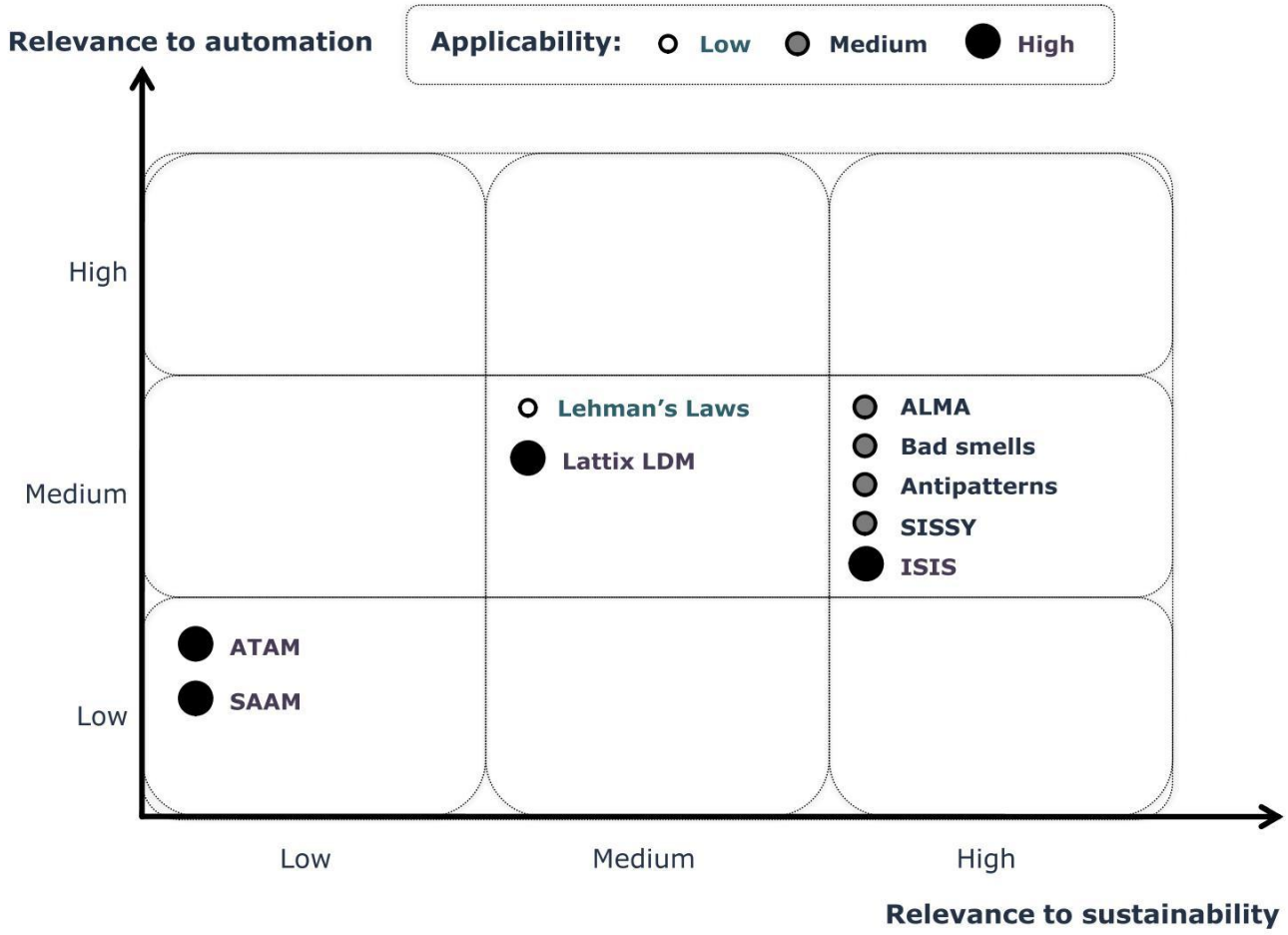

Figure 3.2: Evaluation overview of approaches for analysis of evolution problems

Here the approaches are placed in the matrix according to their relevance for automation domain and sustainability. Relevance is being estimated by handling one or several of the following criteria:

- applicable to embedded systems
- considers concurrency / safety
- respects expensive hardware / environment
- failures / safety issue become very expensive
- large investments implied by each installation: investments must be assured

The thickness of the point indicates their applicability. The applicability is being estimated using the following criteria:

- C/C++ / legacy code support
- available tooling
- tooling tested in case studies
- tooling tested in industrial studies

- support for tooling available

A thicker point indicates a higher applicability.

3.1 Lehman's Laws

Goal: Serve as a base for understanding "aging" processes of software systems. Such understanding may enable to optimize design and development of a system in order to minimize maintenance and evolution overhead in future. Lehman's Laws are listed in this review in order to provide an introduction to the solution strategies to evolution problems.

Short Description: A fundamental article by Lehman (1980) divides Software into three categories, depending on their evolution behaviour:

- S-Systems (Specifiable) - stable and do not evolve
- P-Systems (Problem solving) - do evolve
- E-Systems (Embedded in concrete context) - do evolve and cause their own evolution

The Lehman's Laws, or so-called Laws of Software Evolution, were formulated by Lehman and Belady starting from 1972 during their research of evolution history of big software systems. The laws are believed to apply mainly to monolithic, proprietary software, however some empirical observations coming from the study of open source software development appear to challenge some of the laws [MFRD08, WYL08, XCI09].

The laws are provided in the table below and can be summarized as follows:

- Law of continuing change: A system that is being used undergoes continuing change or degrades in effectiveness.
- Law of increasing complexity: A computer program that is changed, becomes less and less structured. The changes increase the entropy and complexity of the program.

The initial system development influence is not reflected in the laws, although it impacts further maintenance and system evolution stages.

Nr. (Date)	**The Law**	**Explanation**
1 (1974)	Continuing Change	E-type systems must be continually adapted or they become progressively less satisfactory.
2 (1974)	Increasing Complexity	As an E-type system evolves, the complexity of its structure increases unless work is done to maintain or reduce it.
3 (1974)	Self Regulation	An E-type system evolution process is self regulating with a distribution of product and process measures close to a normal distribution.
4 (1978)	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
5 (1978)	Conservation of Familiarity	As an E-type system evolves, anything associated or working with it (for example developers, sales personnel or users) must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

6 (1991)	Continuing Growth	The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
7 (1996)	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
8 (1996)	Feedback System (first stated 1974, formalized as law 1996)	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Evaluation:

Development Phase:	Design
Relevance, Automation:	Medium (Not automation specific, no limitation for domain)
Relevance, Sustainability:	Medium (Provide generalisation and basic understanding, however laws do not hold for all systems)
Applicability:	Low (More general principles)
Tool:	No tool support available. Lehman's Laws represent a rule of thumb.
Preventive / Reactive / Analytical:	Preventive
Formalization:	Informal
Perspective:	Positive
Abstraction level:	High
Benefit for sustainable systems:	Help to understand maintenance and evolution phases, and, therefore, consider them during system development. This may help to minimise future overhead.

References:

- [C. Bommer, M. Spindler, and V. Barr. Software Wartung. dpunkt.verlag, 2008. Chapter 4.2] [BSB08] (book)
- Wikipedia: http://en.wikipedia.org/wiki/Software_evolution
- [T. Mens, J. Fernandez-Ramil, and S. Degrandsart. The evolution of Eclipse. IEEE International Conference on Software Maintenance, 2008. ICSM 2008. pages 386 – 395, 2008] Presents a metrics-based study of the evolution of Eclipse, an open source integrated development environment, based on data from seven major releases, from releases 1.0 to 3.3. Investigates whether the laws of software evolution were supported by the data. The authors found supportive, as well as contradictive evidences. [MFRD08] (conference paper)
- [M. Wermelinger, Yijun Yu, and A. Lozano. Design principles in architectural evolution: A case study. IEEE International Conference on Software Maintenance, 2008. ICSM 2008. pages 396 – 405, 2008] Investigates how structural design principles are used in practice, in order to assess the utility and relevance of such principles to the maintenance of large, complex, sustainable and successful systems. Eclipse is used for the case study in order to check whether its architecture follows, throughout multiple releases, some proposed evolution principles. The authors found supportive as well as contradictive evidences. [WYL08] (conference paper)
- [Guowu Xie, Jianbo Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. IEEE International Conference on Software Maintenance, 2009. ICSM 2009. pages 51 – 60, 2009] Tries to verify Lehman's laws of software evolution. The findings indicate that several of these laws are confirmed, while the rest can be either confirmed or disproved depending on the law's operational definitions. Analyses the growth rate for projects development and maintenance branches, and the distribution of software changes. Finds similarities in the evolution patterns of the studied programs (Plan: construction of rigorous models for software evolution). [XCI09] (conference paper)
- [A. Israeli, D. G. Feitelson. The Linux kernel as a case study in software evolution. Journal of Systems and Software, Volume 83 , Issue 3, pages 485-501, 2010] [IF10] (journal paper)

3.2 Architecture-Based Understanding and Description

Software Architecture plays an important role in Software Engineering. In this chapter we present approaches which use software architecture modelling for analyses and description of evolution problems.

There are several surveys of scenario-based architecture analysis approaches:

- [M.A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. Australian Software Engineering Conference 2004. Proceedings, 2004, pages 309–318, 2004] A Framework for Classifying and Comparing Software Architecture Evaluation Methods, Covers: Software Architecture Analysis Method (SAAM), Architecture Tradeoff Analysis Method (ATAM), Active Reviews for Intermediate Design (ARID), SAAM for Evolution and Reusability (SAAMER), Architecture-Level Modifiability Analysis (ALMA), Architecture-Level Prediction of Software Maintenance (ALPSM), Scenario-Based Architecture Reengineering (SBAR), SAAM for Complex Scenarios (SAAMCS), and integrating SAAM in domain-Centric and Reuse-based development (ISAAMCR) [BZJ04]. (conference paper)

- [E. Dobrica, L. Niemela. A survey on software architecture analysis methods. Transactions on Software Engineering, 28(7): 638–653, 2002] A Survey on Software Architecture Analysis Methods, Covers: software architecture analysis method (SAAM) and its three particular cases of extensions, one founded on complex scenarios (SAAMCS), and two extensions for reusability, ESAAMI and SAAMER, the architecture trade-off analysis method (ATAM), scenario-based architecture reengineering (SBAR), architecture level prediction of software maintenance (ALPSM), and a software architecture evaluation model (SAEM) [Dob02]. (conference paper)

3.2.1 Approach: Architecture Tradeoff Analysis Method, ATAM

Goal: Support an architectural decision process considering a tradeoff between several quality attributes.

Short Description: A structured approach for understanding the tradeoffs inherent in the architectures of software-intensive systems. This approach was developed to provide a principle way to evaluate a software architecture's fitness with respect to multiple competing quality attributes: modifiability, security, performance, availability, and so forth. The approach helps to reason about architectural decisions that affect quality attribute interactions, e.g. when improving one quality attribute leads to decreasing another.

The ATAM approach comprises nine steps which are structured in four groups: Presentation (Present the ATAM, Present business drivers, Present architecture), Investigation and Analysis (Identify architectural approaches, Generate quality attribute utility tree, Analyze architectural approaches), Testing (Brainstorm and prioritize scenarios, Analyze architectural approaches), and Reporting (Present results).

Architectural approaches are architecture styles, or in general architectural design decisions, that should be evaluated. A utility tree is used to elicit the quality attributes of interest and determine a ranking of quality attributes.

ATAM consists of four phases, which are performed in a couple of stakeholder workshops. Phase 0 is a setup phase. Phase 1 and 2 are the evaluation phases and comprise the nine steps mentioned before. Phase 1 is architecture-centric with focus on eliciting architectural information. Phase 2 is stakeholder-centric and aims at eliciting stakeholder points of view. Phase 3 is a follow-up phase for producing a final report and planning follow-on actions.

The approach has been applied to several case studies and software systems in practice.

Evaluation:

Development Phase:	Design / Maintenance
Relevance, Automation:	Low (Approach is not addressing automation itself, depending on participating stakeholders)
Relevance, Sustainability:	Low (Approach is not addressing sustainability itself, depending on participating stakeholders)
Applicability:	High (Can be applied without tools by performing workshops with stakeholders)
Tool:	There is a journal paper about a collaborative web-based architecture evaluation tool for ATAM, however this tool is not available in Web [MT05].
Preventive / Reactive / Analytical:	Analytical
Formalization:	Informal (Manual work, high stakeholder involvement)
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Applying ATAM with focus in sustainability might help stakeholders to get a better common understanding of sustainability factors and improve decisions, but depending on stakeholder's experience.

References:

- [P. Clements, R. Kazman, and M. Klein. Evaluating software architectures. Addison-Wesley, 4. print. edition, 2005] [CKK05] (book)
- [R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. Software, IEEE, 13(6), pages 47 – 55, 1996] [KKB+98] (conference paper)
- [Piyush Maheshwari and Albert Teoh. Supporting atam with a collaborative webbased software architecture evaluation tool. Science of Computer Programming, 57, pages 109 – 128, 2005] [MT05] (journal

paper)

- Official Webpage: http://www.sei.cmu.edu/architecture/tools/atam/

3.2.2 Approach: Software Architecture Analysis Method, SAAM

Goal: Analyse software architecture and get a better understanding of sustainability problems.

Short Description: An approach for analyzing software architectures by using a set of scenarios. Based on the observation that claims about qualities, like "a certain pattern ensures maintainability of the system", could not be tested effectively, the creators of SAAM replaced such claims with scenarios that operationalize those claims. Originally SAAM was focused on modifiability analyses, but has proven to be useful for many other quality attributes as well (such as portability, extensibility, etc.). Stakeholders enumerate a set of scenarios that are scrutinized, prioritized, and mapped onto a representation of the architecture.

Scenarios are brief descriptions of some anticipated or desired use of a system. A scenario that is directly supported by an architecture is called *direct scenario*. A scenario that requires modifications of the architecture is called *indirect scenario*. The identification of indirect scenarios is an important objective of SAAM.

SAAM comprises of the following steps: 1) develop scenarios, 2) describe architecture, 3) classify and prioritize scenarios, 4) individually evaluate indirect scenarios, 5) assess scenario interactions, 6) create overall evaluation.

Evaluation:

Development Phase:	Design / Maintenance
Relevance, Automation:	Low (Approach is not addressing automation itself, depending on participating stakeholders)
Relevance, Sustainability:	Low (Approach is not addressing sustainability itself, depending on participating stakeholders)
Applicability:	High (Can be applied without tools by performing workshops with stakeholders)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Helps stakeholders to get a better understanding of sustainability problems when applied with focus on evolution, but depending on stakeholder's experience.

References:

- [P. Clements, R. Kazman, and M. Klein. Evaluating software architectures. Addison-Wesley, 4. print. edition, 2005] [CKK05] (book)
- [R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. Software, IEEE, 13(6):47 – 55, 1996] [KKB+98] (conference paper)

3.2.3 Approach: Architecture-Level Modifiability Analysis, ALMA

Goal: Besides other, analyse change efforts and identify evolution risks.

Short Description: Architecture-level modifiability analysis (ALMA), a architecture-level analysis approach that focuses on modifiability, distinguishes multiple analysis goals, has explicit assumptions and provides repeatable techniques for performing the steps.

ALMA consists of five main steps, i.e. goal selection, software architecture description, change scenario elicitation, change scenario evaluation, and interpretation. Supported goals are: maintenance effort prediction, risk assessment, and comparison of candidate architectures.

For description of the architecture an architectural model, i.e., views from several architectural viewpoints have to be created. The kind of model is not fixed. Change scenario elicitation is done by interviewing of stakeholders. Since the number of possible changes is almost infinite the approach considers usage of equivalence classes and classification of change categories for change scenarios. The authors propose two techniques for selection of scenarios: 1) top-down: predefined classification of change categories is used to guide the search for change scenarios, 2) bottom-up: stakeholders are interviewed without predefined classification. The scenario evaluation step evaluates the effect of the change scenarios on the architecture. Therefore three impact analysis steps are performed: 1) Identify affected components, 2) Determine effect on the components, 3) Determine ripple effects.

The approach has been validated through its application in several cases, including software architectures at Ericsson Software Technology, DFDS Fraktarna, Althin Medical, the Dutch Department of Defense and the Dutch Tax and Customs Administration.

Evaluation:

Development Phase:	Design / Maintenance
Relevance, Automation:	Medium (There is no limitation of domain)
Relevance, Sustainability:	High (Approach addresses modifiability, change effort explicitly)
Applicability:	Medium (Manual approach, modelling is required but no tool support available, difficult to rate since there a only a few case study reports.)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Informal
Perspective:	Positive / Negative
Abstraction level:	Medium / High
Benefit for sustainable systems:	Approach can help identifying evolution risks, i.e. changes that are critical for sustainability and can only be performed at high costs.

References:

- [P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). Journal of Systems and Software, 69(1-2):129 – 147, 2004] [BLBvV04] (journal paper)

3.3 Software Comprehension by using Data Mining

An investigation of project history gives insights into the evolution process of a system. Approaches belonging to this category use data from the (past) projects (e.g. code, bug tracker systems or archived project communication) and data mining techniques (e.g. generalization or clustering) in order to derive knowledge about evolution stages, frequent causes and problems.

Due to space restrictions and presence of an excessive survey ([80309]), this section only highlights general aspects, which are representing a general introduction to the topic.

Data mining has two general purposes:

- To uncover hidden dependencies in the data ("descriptive")
- To predict possible future development scenarios of the data ("predictive")

An overview of the data mining techniques, goals (sections) and information sources is presented in Figure 3.3 ([Has08]).

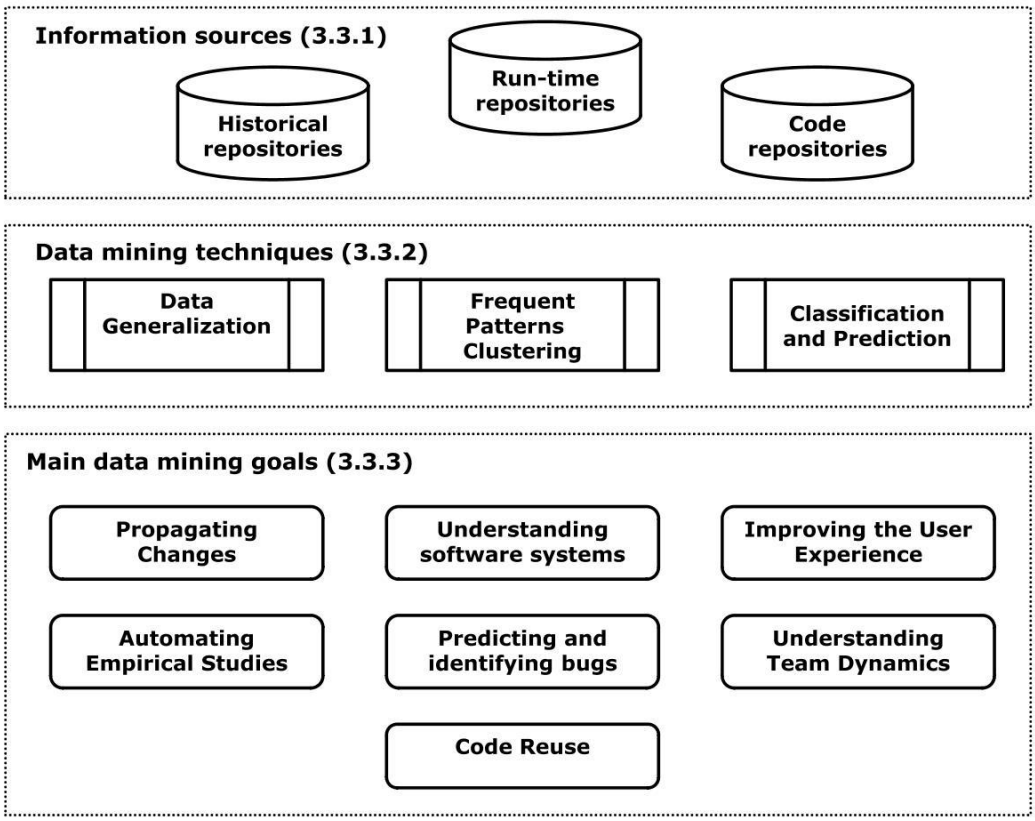

Figure 3.3: Overview: Data mining research sections (goals), approaches and information sources

3.3.1 Information sources

An important part of data mining is comprehensive data, which plays the base for the data mining techniques. Such techniques are used to process and analyse different data types from various information sources depending on the goals of such analysis (see section above). Examples of data information sources (both software and document repositories)[Has08] (journal paper) are:

- Historical repositories, such as source control repositories, bug repositories, and archived communications, record informations about the evolution and progress of a project.

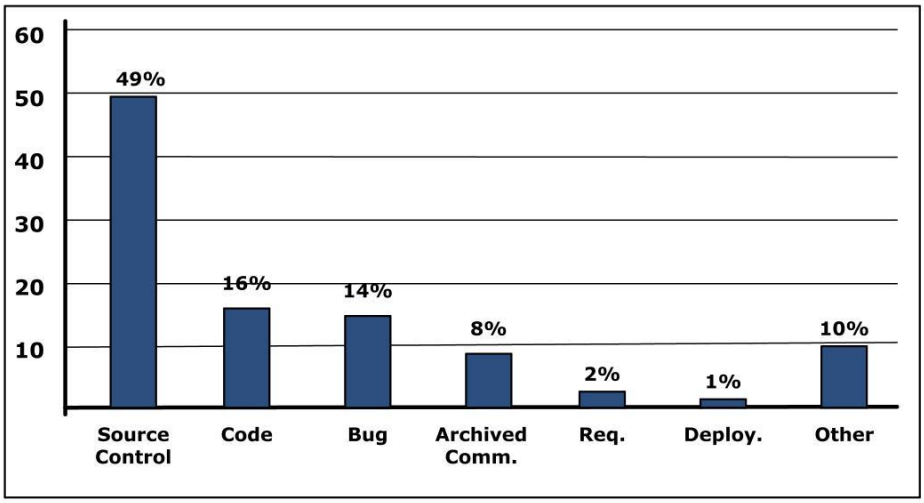

Figure 3.4: Repositories in data mining, publications 2004-2008 ([Has08]) (journal paper)

- Run-time repositories, such as deployment logs (e.g. logs of software or updates installed by different user groups, Web services logging), contain information about the execution and the usage of an application at a single or multiple deployment sites.
- Code repositories, such as Sourceforge.net and Google code, contain the source code of various applications developed by several developers.

Mining these historical, run-time and code repositories, one can uncover patterns and information, e.g. historical dependencies between project artefacts, such as functions, documentation files, or configuration files. Such historical dependencies may be a link between an update of a requirement document and introduction of a new feature into the system (e.g. discovered by corresponding changes in code). One can also eventually trace this change propagation through several system parts and therefore, for example, predict which system parts will be sequentially affected by maintenance activities. Such pattern and information analysis runs (semi-) automatically.

The study provided by [Has08] analyses the publications made in the 2004-2008 along various types of repositories (see Figure 3.4). It shows that a large amount (more than 80%) of the published work focuses on source code and bug related repositories. The Figure counts all types of repositories used by a particular paper.

3.3.2 Data mining techniques

A detailed overview of currently used data mining techniques is presented in [80309] (project survey). It reviews data types, data processing, most common data mining techniques, mining complex data sets and future trends in the data mining field.

According to [80309] (project survey), data mining techniques can be divided into the following groups:

- Data generalisation - is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level, such as numerical values, to higher conceptual levels, such as categorical labels, in concise terms, at different granularity, and from different angles. Data generalization is a form of descriptive data mining that discovers interesting general properties of the data.
- Frequent pattern clustering - is the process of grouping the data into classes or clusters so that objects within a cluster have high similarity to one another within the same cluster

but are dissimilar to objects in other clusters. The quality of clustering can be assessed by a measure of dissimilarity between objects. Frequent patterns are substructures, subsequences, or item sets (a set of items) that appear in a data set frequently. One example of it is that milk and bread may frequently appear together in a shopping transaction data set.

- Classification and prediction - are two forms of data analysis to extract models describing data classes or to predict future data trends. Classification predicts categorical labels, whereas prediction models continuous-valued functions.

For further information please refer to the former mentioned document.

3.3.3 Data mining goals

The main directions of the data mining goals are listed in the following [Has08] (journal paper). Since [Has08] represents an extensive overview, only the main research directions are listed. For further details refer to the overview article.

- **Understanding (large) software systems.** Large software systems are often insufficiently documented or/and the documentation is out of date. The initial project participants are as well usually no longer available. Dependency graphs and the source code documentation offer a static view of a system and fail to reveal details about the history of a system or the rationale for its current state or design. Sustainable systems typically suffer from outdated documentation (respectively, such systems are usually under-documented), thus this data mining goal is of high interest for sustainable systems.

- **Propagating Changes.** Current practices for propagating software changes often rely on human communication, and the knowledge of experienced developers. Many bugs that are hard to find are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly. However, instead of using traditional dependency graphs to propagate changes, one could make use of historical co-changes. The assumption here is that entities co-changing frequently in the past are very likely to co-change in the future. This data mining research direction is of high interest for sustainable systems.

- **Automating Empirical Studies.** The automation of empirical studies permits the repetition of studies on a large number of subject and the ability to verify the generality of many findings in these studies. This goal is less relevant for the goals of this project, as it provides more theoretical result having less immediate impact on evolution.

- **Understanding Team Dynamics.** Many large projects communicate through mailing lists, IRC channels, or instant messaging. These discussions cover many topics such as future plans, design decisions, project policies, and code or patch reviews. Is less relevant for the goals of this project, as a suitable information for such analysis first has to be gathered that might be conflicting with information and privacy protection policies.

- **Predicting and identifying bugs.** The assumption is that the code that had bugs in the past is likely to have bugs in the future. Furthermore, bugs are not likely to appear in unchanged code. Thus, recent bugs and changes have a higher effect on the bug potential of a code over older changes. Although, early bug prediction and identification could improve overall maintenance effort, the efficiency of such predictions would first have to be investigated. Therefore, this goal is less relevant for the goals of this project.

- **Improving the User Experience.** One can mine reported bugs and execution logs to prevent an application from crashing, by warning the user, when he attempts to perform an action which has been reported by others to have bugs. However, in the automation

domain a special attention to the system's robustness. Such systems are not allowed to crash on user actions, therefore, this direction is less relevant for the goals of this project.

- **Code Reuse.** Large code libraries are usually not well-documented, and have complex APIs, which are hard to use by non-experts. So one could mine code repositories to help developers reuse code. However, the extend of reuse of such code libraries by ABB is unclear. Additionally, tools required to support such analysis and their efficiency will first have to be implemented and evaluated. Therefore, this direction is less relevant for the goals of this project.

Further on, data mining techniques may consider both architecture level and code level, and may be used during design, implementation and maintenance phases of system life.

Data mining techniques concerned with understanding (large) software systems and propagation of changes are matters of the highest interest for sustainable systems. We provide several example references to the approaches concerned with the mining information repositories for understanding software systems and propagating changes.

References:

- [A.E. Hassan. The road ahead for Mining Software Repositories. Frontiers of Software Maintenance, 2008. FoSM 2008. pages 48 – 57, 2008] [Has08] (journal paper)
- [CRID 80321. Data Mining Survey, 200] [80309] (project survey, CRID)
- [T. Girba and S. Ducasse. Modeling History to Analyze Software Evolution. Journal of Software Maintenance: Research and Practice (JSME), pages 207 – 236, 2006] Current approaches reasoning about general laws of software evolution do not rely on an explicit metamodel, and thus, they make it difficult to reuse or compare their results. The authors present a survey of the evolution analyses and deduce a set of requirements that an evolution meta-model should have. They define, Hismo, a meta-model in which history is modelled as an explicit entity: it adds a time layer on top of structural information, and provides a common infrastructure for expressing and combining evolution analyses and structural analyses. The authors validate the usefulness of their meta-model by presenting how different analyses are expressed with it [GD06]. (journal paper)
- [A.E. Hassan, Zhen Ming Jiang, and R.C. Holt. Source versus object code extraction for recovering software architecture. 12th Working Conference on Reverse Engineering, page 10, 2005] To support developers maintaining and evolving under (or un-) documented systems, an up to date view of the architecture could be recovered from the system's implementation. Source code or object code extractors may be used to recover the architecture. This paper explores using two types of extractors (source code and object code extractors) to recover the architecture of several large open source systems. It investigates the differences between the results produced by these extractors to gain a better understanding of the benefits and limitations of each type of extractor. Results show that both types of extractors have their benefits and limitations. For example, an object code extractor is easier to implement while a source code extractor recovers more dependencies that exist in the source code as seen by developers [HJH05]. (conference paper)
- [L. Moonen, PhD thesis, The University of Amsterdam. Exploring Software Systems. 2002] In the dissertation, the author investigates approaches and tools that help remedy the knowledge degradation about a system during its evolution by supporting the exploration of a software system and improving its legibility. He examines an analogy with urban exploration and present approaches for the extraction, abstraction, and presentation of information needed for understanding software [Moo02]. (PhD thesis)
- [L. Canfora, G.and Cerulo and M. Di Penta. Tracking Your Changes: A Language-Independent Approach. Software, IEEE, 26 , Issue:1, 50 –57, 2009] The authors use versioning systems (svn, etc.) for studying and monitoring a software system's evolution. The approach enables tracking the evolution of software entities treatable as a sequence of lines, such as source code, but also requirements, use cases, and test cases [CDP09] (journal paper)
- [D. Cubranic. Project History as a Group Memory: Learning From the Past. PhD thesis, 2005] Proposes a tool called Hipikat which indexes historical repositories, and displays on demand relevant historical information (Email exchange, bug reports, etc.) within the development environment. Hipikat then recommends relevant artefacts from the memory during a software modification task. While working on a particular change, Hipikat can display pertinent artefacts such as old emails and bug reports, discussing the code being viewed in the code editor [Cub05]. (PhD thesis)
- [A.E. Hassan and R.C. Holt. Using development history sticky notes to understand software architecture. Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on, pages 183 – 192, 2004] An approach that recovers information (called source sticky notes) from source control systems and attaches this information to the static dependency graph of a software system. The article demonstrates

how to use these notes along with the software reflexion framework (has been proposed by Murphy et al. to assist in understanding the structure of software systems, [MNS95]) to assist in understanding the architecture of large software systems. These notes record various properties concerning a dependency such as the time it was introduced, the name of the developer who introduced it, and the rationale for adding it. NetBSD (an open source operating system) is used as an example for demonstration [HH04]. (conference paper)

- [A.E. Hassan. Mining software repositories to assist developers and support managers. In Proceedings of ICSM 2006: IEEE International Conference on Software Maintenance, Chicago, Philadelphia, USA, pages pp. 339–342, Sept. 24-27, 2006] This paper presents approaches and tools which mine and transform static record keeping software repositories to active repositories. Such repositories then can be used by researchers to gain empirically based understanding of software development, and by practitioners to predict, plan and understand various aspects of their project [Has06]. (conference paper)
- [M. Di Penta. Evolution Doctor: A Framework to Control the Evolution of Undocumented Software Systems. PhD thesis, 2003] This work proposes a framework, Evolution doctor, to diagnose and "cure" software erosion caused by the maintenance activities. Some example of negative maintenance activities (mentioned in the work) are growth of cloning percentage, the presence of unused objects, the loss of source file organisation and traceability links [Pen03]. (PhD thesis)
- [D. Poshyvanyk. Using information retrieval to support software maintenance tasks. ICSM 2009. IEEE International Conference on Software Maintenance, 2009, pages 453 – 456, 2009] Defines and validates a semi-automated approach for feature location in source code based on the combination of a single execution trace and comments and identifiers from source code. The approach is based on information retrieval (IR) approach for extracting and representing the unstructured information in large software systems. These approaches are supposed to contribute directly to the improvement of design of incremental changes and thus increased software quality and reduction of software maintenance costs [Pos09]. (conference paper)
- [L. Cerulo. On the Use of Process Trails to Understand Software Development (Analysis of historical data in combination with static and dynamic analysis). PhD thesis, 2006] The author investigates the usefulness of historical data stored in software repositories to support developers and managers in maintenance activities. There are three case studies: 1) derivation of the set of files impacted by a change by considering those that have been impacted by similar changes in the past; 2) selection of the best developers able to resolve a new change request; 3) derivation of presence of crosscutting concerns in source code, the hypothesis is that developers usually perform logical transactions coupled in a concern. The approach is implemented in an Eclipse plug-in: Jimpa [Cer06]. (PhD thesis)
- [A. Tarvo. Mining Software History to Improve Software Maintenance Quality: A Case Study. Software, IEEE, Volume: 26 , Issue: 1, 34 – 40, 2009] A case study on the "Binary Change Tracer", which collects data on software projects and helps predict regressions (e.g. decreases of quality through introduction of a new feature) in software projects. Microsoft and C++ based [Tar09]. (journal paper)

3.4 Monitoring and Evaluating Quality Indicators

Quality Indicators help to identify potential problems in the software, which may have an impact on system evolution.

- **Quantitative Indicators** are numeric representations of software quality properties and have to be interpreted in order to derive the semantic rationale of quality problems. Software metrics are typical representatives of this category.
- **Qualitative Indicators** give hints on potential software quality problems. Qualitative indicators can provide the semantic rationale of software quality problems. Typical representatives of this category are e.g., antipatterns. An antipattern is associated with an explicit knowledge about negative quality impact, i.e., the semantic rationale.

3.4.1 Summary: Metrics for Identifying Evolution Problems

A software metric is a measure of some property of a software or its development process. Metrics help analyse the state, quality and (possible) troubles of a system. They can be used to support the implementation, maintenance and evolution phases. Furthermore, they can be interpreted as quantitative software quality indicators.

Metrics belong to quality assurance activities and can be divided into the three general categories:

- Product metrics - characteristics of the product such as size, complexity, design features, performance, and quality level. Can be divided into conventional metrics and object-oriented metrics. These can be further dived on size metrics (e.g. amount of entities) and structure / complexity metrics (e.g. coupling, cohesion).

  Conventional size metrics are for example LOC (lines of code), NOS (number of statements) and Halstead (determines a quantitative measure of complexity directly from the operators and operands in the module, can be used for the estimation of maintenance and test overhead).

  Conventional structural metrics are for example Cyclomatic complexity (measures the number of linearly independent paths through a program's source code), Fan-in / Fan-out (measure the relationships/calls between procedures/modules) and Maintainability index (is calculated from lines-of-code measures, McCabe measures and Halstead metrics with coefficients, can be used for maintainability evaluation and monitoring).

  Object-oriented size metrics are for example NOA (number of attributes per class)and NOM (number of methods per class). They can be used for estimation of a maintainability overhead.

  Object-oriented structural metrics are for example CBO (coupling between objects), DIT (depth inheritance tree) and NOC (number of children).

- Process metrics - can be used to improve software development and maintenance, e.g. effectiveness of defect removal, response time of the fix process, and pattern of testing defect arrival. Process metrics can help answering the following questions, considering evolution of the systems. Such questions can be for example about size of old / new release, how often is system change needed or how much does a change cost.

  An example of a process metric: Growth of the system = (Size of the new release - Size of the old release) / Size of the old release.

- Project metrics - describe the project characteristics and execution, e.g. the number of software developers, the stuffing pattern over the life cycle of software, cost, schedule, and productivity.

Such metrics can be used for the project monitoring. However they are of less practical interest for the prevention evolution problems. Metrics may help to monitor development process and issues influencing later maintenance, evolution and migration activities.

Tools like SISSY, Section 3.4.3.2, and ISIS, Section 3.4.3.3, are designed to execute automatic analyses and metric calculation so that the potential system's problems are uncovered. Another tool suite for static code analysis and metric calculation is the Sotograph Software-Tomograph, Sotograph.

Some metrics may belong to multiple categories. Metrics help to analyse and monitor the system and thus indicate the potential problems. This contributes to the general quality and understandability, and lowers complexity of a system.

References:

- [S. H. Kan. Metrics and Models in Software Quality Engineering (2nd ed.). Addison-Wesley Longman, 2002] [Kan02] (book)
- [C. Bommer, M. Spindler, and V. Barr. Softwarewartung. dpunkt.verlag, 2008. Chapter 9] [BSB08] (book)
- [N. E. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. PWS Publishing Company, 1997] (book) [FP97] (book)
- [B. Henderson-Sellers. Object-oriented Metrics Measures of Complexity. Prentice Hall, 1996] [HS96] (book)
- [R. Dumke and F. Lehner. Software-Metriken Entwicklungen, Werkzeuge und Anwendungsverfahren. Deutscher Unversitaets-Verlag, 2000] [DL00] (book)
- [S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, v. 20, 476 – 493, 1994] [CK94] (journal paper)
- [M. Alshayeb and Wei Li. An empirical validation of object-oriented metrics in two different iterative software processes. IEEE Transactions on Software Engineering, v. 29, Issue: 11, 1043 – 1049, 2003] [AL03] (journal paper)
- SISSy project, Section 3.4.3.2: http://sissy.fzi.de
- ISIS project, Section 3.4.3.3: http://www.andrena.de/node/160/
- Sotograph: http://www.software-tomography.ch/html/sotograph.html

3.4.2 Approach: Detection of Bad Smells or Antipatterns

Goal: Detect possible problems at the code level.

Short Description: A bad smell or antipattern is any symptom in the source code or design that possibly indicates a deeper problem. It might not be an error, but a situation that might easily lead to an error when changing the system.

Examples for bad smells in code (according to [FBB+99](book)) are:

- Duplicated Code: changes to duplicated code might lead to inconsistency and adaptation overhead since several locations need to be adapted instead of one.
- Shotgun Surgery: Elements which belong together (and are usually changed together) are spread over a lot of different classes and locations. When changes are all over the place, they are hard to find, and it's easy to miss an important change.
- Divergent Change: If a class is commonly changed in different ways for different reasons. This indicates an suboptimal encapsulation structure.

There are formalised heuristics for detecting bad smells in code and appropriate tools support, e.g. SISSy, FindBugs, etc.

[RL04] (book) provides a small collection of architectural smells. This comprises smells regarding packages, subsystems, and layers.

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (Not automation specific, no limitation for domain)
Relevance, Sustainability:	High
Applicability:	Medium (Approaches highlight potential quality problems, which must be tackled manually.)
Tool:	Many (e.g. SISSY Problem pattern detection (see Section 3.4.3.2), FindBugs)
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal (Formalized heuristics)
Perspective:	Negative
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Detect critical areas in systems, which block or impair evolution.

References:

- [M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999, Chapter 3, Bad Smells in Code] [FBB+99] (book)
- [F. Simon, O. Seng, and T. Mohaupt. Code-Quality-Management. dpunkt.verlag, 2006, Chapter 10, Catalog of Quality Indicators] [SSM06] (book)
- [S. Roock, M. Lippert. Refactorings in grossen Softwareprojekten. Dpunkt-Verlag, 1. au. edition, 2004] [RL04] (book)
- SISSy project, Section 3.4.3.2: http://sissy.fzi.de
- FindBugs: http://findbugs.sourceforge.net/

3.4.3 Architectural Enforcements

3.4.3.1 Approach: Dependency-Analysis using Lattix LDM

Goal: Find design rule and architectural violations in Java / C++ code via graphical visualization in so-called dependency structure matrices.

Short Description: The Lattix LDM tool takes Java or C++ files as inputs and performs a static code analysis. The result is a dependency structure matrix (DSM) visualizing the call dependencies between classes or packages. It reflects the hierarchical structure of the code. Using the DSMs layering violation in the code can be detected relatively easily therefore giving pointers to potential architectural erosion. The code has to be refactored manually to remove these violations. The tool helps to check the compliance of code with architectural principles. Code adhering to a high-level structure is supposed to be easier maintainable and can therefore increase the sustainability of a system.

Evaluation:

Development Phase:	Maintenance
Relevance, Automation:	Medium (Not automation specific, no limitation for domain)
Relevance, Sustainability:	Medium (Detects violations, therefore supports sustainability. Manual actions required)
Applicability:	High (Has been applied in industry)
Tool:	http://www.lattix.com
Preventive / Reactive / Analytical:	Reactive
Formalization:	Not applicable
Perspective:	Negative
Abstraction level:	Low
Benefit for sustainable systems:	Uncover architecture erosion in source code and improve the inner structure of systems to reduce future maintenance efforts

References:

- [N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. Proceedings of OOPSLA'05, pages 167–76, 2005] [SJSJ05] (conference paper)
- [C. Hinsman, N. Sangal, and J. A. Stafford. Achieving Agility through Architecture Visibility. QoSA, Lecture Notes in Computer Science, Springer, Vol. 5581, 116 – 129, 2009] [HSS09] (conference paper)
- URL: http://www.lattix.com

3.4.3.2 Approach: SISSy

Goal: Metric calculation and Problem pattern detection in object-oriented source code.

Short Description: SISSy stands for Structural Investigation of Software Systems. It is a tool for static analysis of object-oriented software. It supports analysis of programs written in Java and C/C++. Programs are extracted into a generalised abstract syntax tree, which is stored into a relational database. SISSy can detect violations of over 50 typical object-oriented design principles, heuristics and patterns, such as bottleneck classes, god classes, data classes, and cyclical dependencies between classes or packages. SISSy is implemented in Java and is available as Eclipse plugin set. It can be installed from sourceforge via Eclipse update site mechanism.

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (As long as object-oriented systems are addressed)
Relevance, Sustainability:	High (See benefits below)
Applicability:	Medium (Automated, already applied in industry in several quality assessments)
Tool:	http://sissy.fzi.de
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Problem patterns usually indicate unnecessary dependencies, too high complexity, bad separation of concerns, etc. The presence of a problem patterns can handicap evolution tasks. Therefore, detection and removal of problem patterns improves sustainability.

References:

- Official Webpage: http://sissy.fzi.de
- Updatesite: http://sissy.sourceforge.net/SISSy_Nightly/
- Q-Bench Webpage: http://www.qbench.de
- [Frank Simon, Olaf Seng, and Thomas Mohaupt. Code-Quality-Management. dpunkt.verlag, 2006] [SSM06] (book)

3.4.3.3 Approach: ISIS

Goal: Navigation system for quality management.

Short Description: ISIS is a tool for detection, scaling of quantitative quality indicators and creation of quality history for development projects. It is developed by andrena objects ag and is used to control the quality of Scrum projects. Core element of ISIS is the *project log*, where indicators (metrics) for process and software quality are automatically recorded, condensed, and documented in time series.

Considered metrics are: Customer satisfaction, Number of bugs, Estimation difference, Average test coverage, Number of Packages in Cycles, Cyclomatic Complexity, Average Component Dependency (on class level), classes with more than 20 methods, methods with more than 15 LOC, and number of compiler warnings. Metrics are condensed into a set of quality indexes, which can be used to visualize internal quality to stakeholders (e.g. managers, customers, etc.). This enables architects to show decrease of software quality, e.g., in case of implementation under time pressure, and to make internal quality aspects explicit in negotiations with management and customers.

The tool is available as RCP application and Eclipse plugin. It is capable to analyse Java and C♯ projects. Metric calculation and static analysis is done with Sotograph tool suite, (http://www.software-tomography.ch). Test coverage is measured using EclEmma, (http://www.eclemma.org/).

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (Focus on object-oriented systems)
Relevance, Sustainability:	High (See benefits below)
Applicability:	High (Tool is used in several industry projects by andrena objects ag)
Tool:	This tool can be provided by andrena object ag
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Keeping code clean and quality index high improves evolution tasks.

References:

- Website about ISIS Project, Used Technology, http://www.andrena.de/projekte/isis
- Website with Project Examples, Contact, Events, http://www.andrena.de/know-how/isis
- Website about Sotograph tool suite, http://www.software-tomography.ch
- Website about EclEmma, http://www.eclemma.org/

Chapter 4

Solving Evolution Problems

Outline

Overview: Approaches pro-actively supporting software evolution can be categorised in several ways. In this chapter we summarise approaches which can be used to actively deal with software evolution during design and implementation of systems. The following strategies provide means to improve the sustainability of software systems.

The classification of solution strategies is presented on the Figures 4.1 and 4.2.

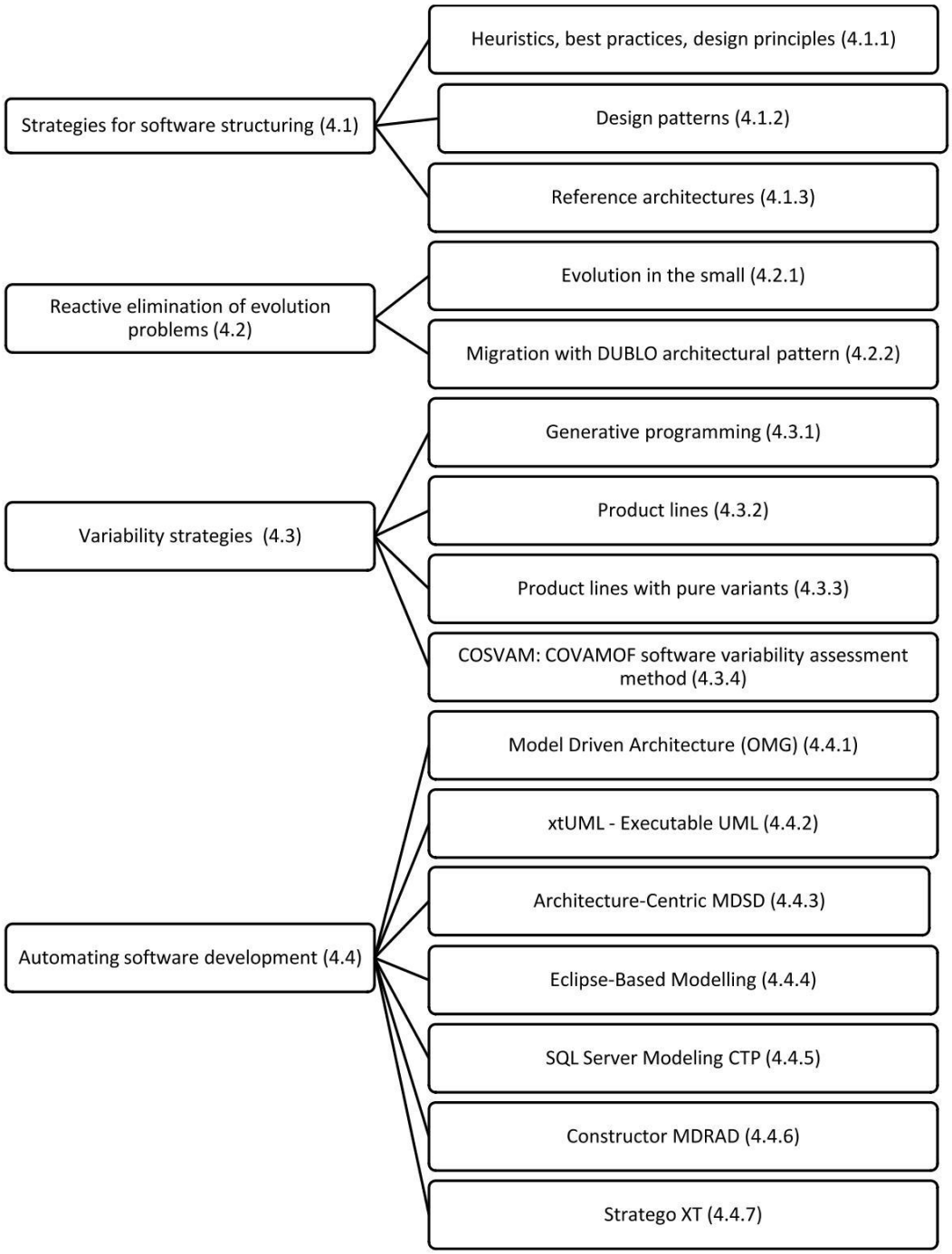

Figure 4.1: Classification of solution strategies, part 1

We present the following types of strategies:

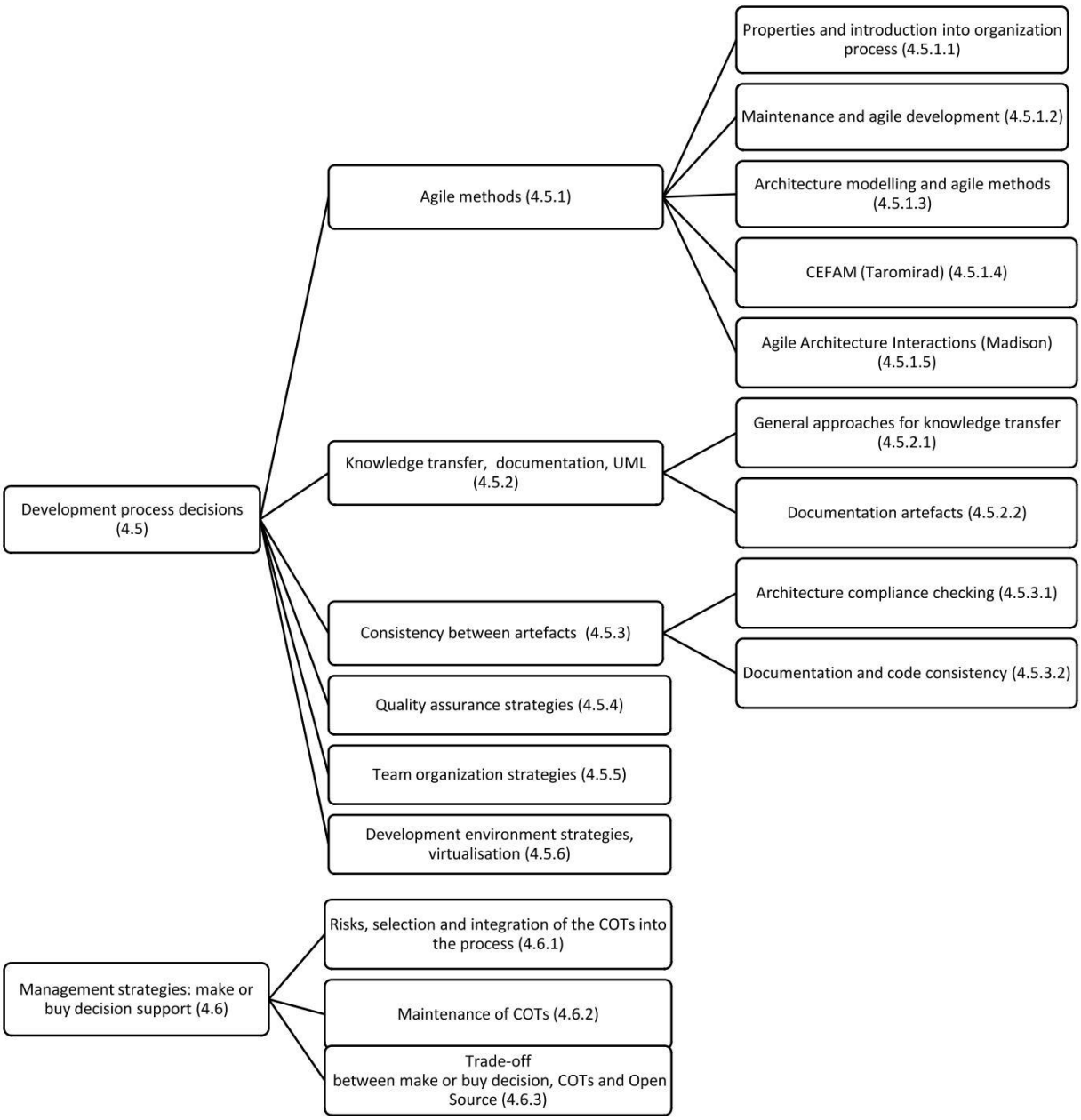

Figure 4.2: Classification of solution strategies, part 2

Strategies for Software Structuring use structural means for reducing complexity and controlling of dependencies. Section 4.1 presents structural strategies in three categories: 1) Design Principles, 2) Reference Architectures, and 3) Patterns.

Reactive elimination of evolution problems deals with approaches that help with systematic improvement of software quality in order to remove evolution problems. In Section 4.2 we present 1) Evolution in the small and 2) Migration with DUBLO architectural pattern.

Variability Strategies apply techniques for domain analysis in order to determine the required variability in software systems. Approaches are presented in Section 4.3 and comprise of 1) Generative Programming and 2) Product Line Approaches.

Automating Software Development describes approaches using modelling in order to increase the abstraction level of software development and automate software development by code generation and model transformations. Section 4.4 covers 1) Model Driven Architecture

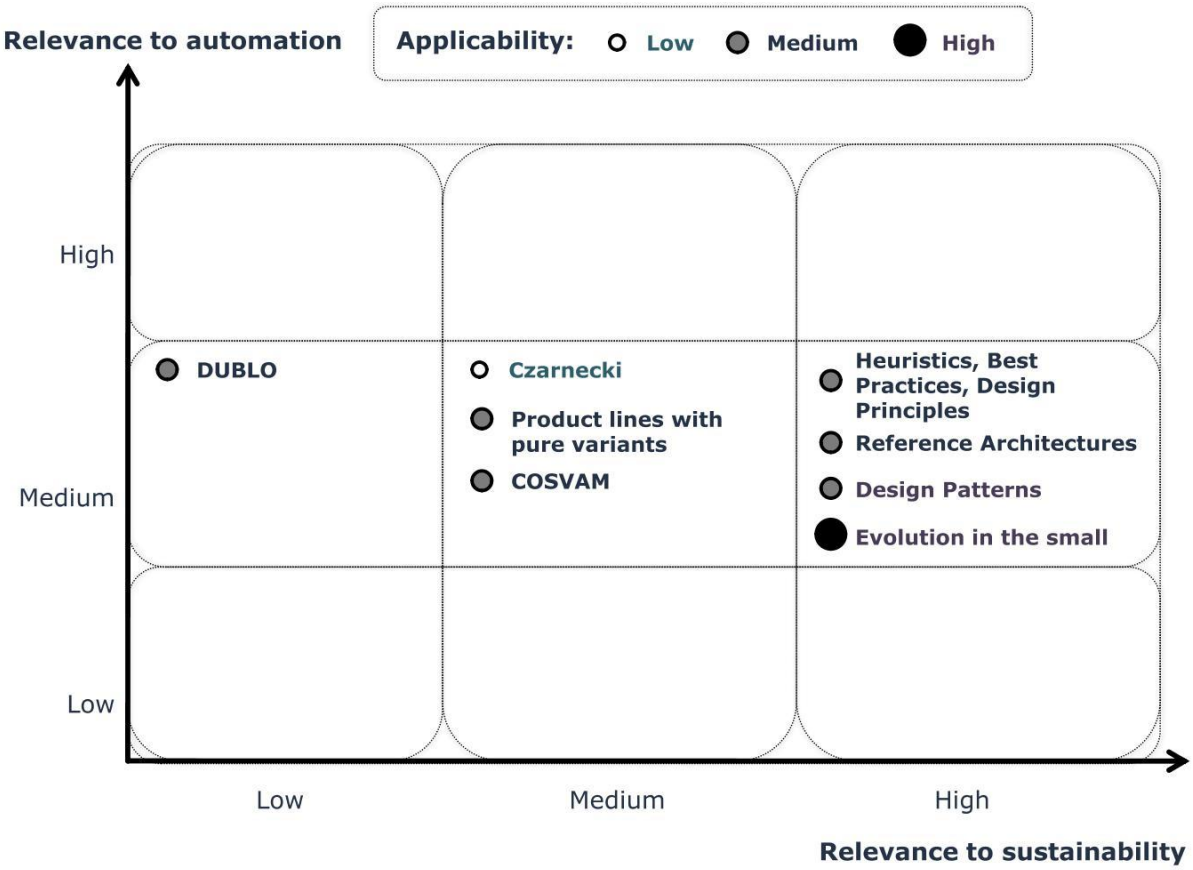

Figure 4.3: Evaluation summary of solution strategies: Strategies for Software Structuring, Variability and Active Elimination

(MDA, following the OMG standard) and 2) Model Driven Software Development (summarising the remainder of model driven approaches).

Development process decisions have influence on sustainability. Therefore we present a selection of approaches in Section 4.5. This comprises strategies dealing with 1) Agile methods, 2) Knowledge transfer, documentation, UML, 3) Consistency of artefacts, 4) Quality Assurance, 5) Team Organization, and 6) Development Environment.

Management Strategies in Section 4.6 deal with the topics of 1) Make or buy decision support and 2) Virtualisation.

Summary sections provide topics descriptions without evaluation and sections dealing with approaches are evaluated according the evaluation criteria. The evaluation results are summarised on Figures 4.3-4.5.

The approaches are placed in the matrix according to their relevance for automation domain and sustainability. The thickness of the point means their applicability (the thicker is the point, the higher is the applicability).

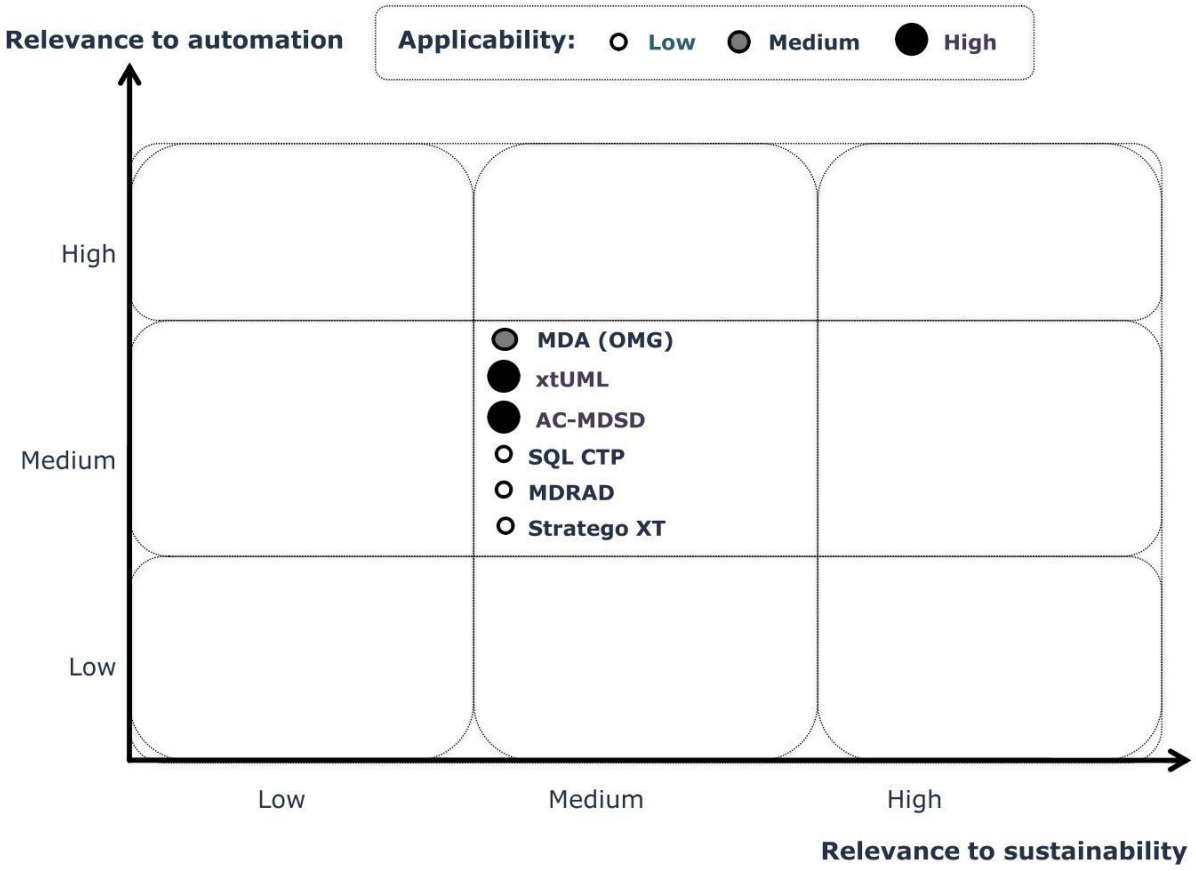

Figure 4.4: Evaluation summary of solution strategies: Automation Strategies

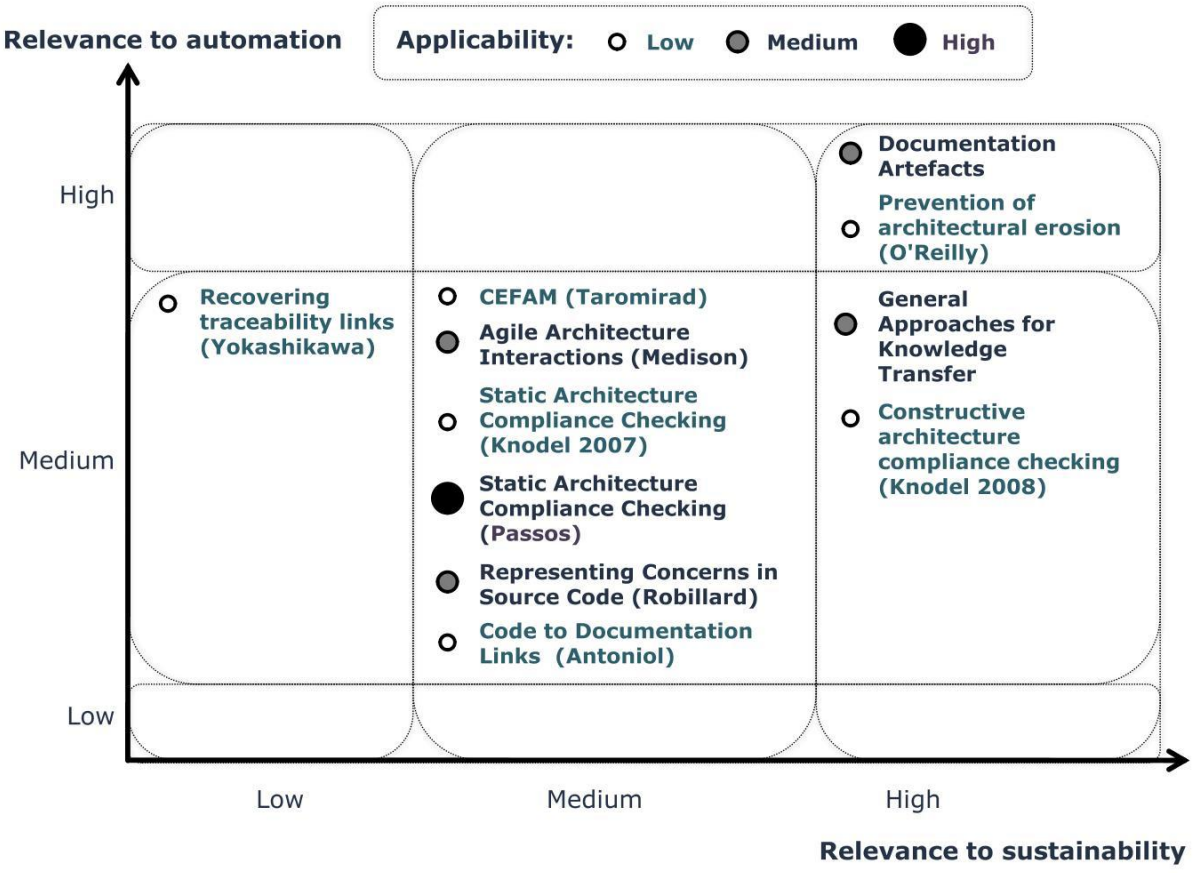

Figure 4.5: Evaluation summary of solution strategies: Development Process Decisions

4.1 Strategies for Software Structuring

In this section we survey approaches that help structuring of software systems in a way, that modifiability and flexibility is increased and sustainability is improved.

4.1.1 Heuristics, Best Practices, Design Principles

Goal: Provide knowledge about good design in an informal way.

Short Description: Best practices cover knowledge of good design in an informal way, predominantly expressed in natural language. They are in principle derived from practical experience and could be expressed either positively (what and how should something be done) or negatively (what should be avoided). Patterns and bad smells can be considered as a kind of formalization of best practices and design principles.

With respect to sustainability there are several best practices and design principles known.

Examples of Object-Oriented Design Principles are:

- The Open Closed Principle (OCP) – A module/component should be open for extension but closed for modification.
- The Liskov Substitution Principle (LSP) – Subclasses should be substitutable for their base classes.
- The Dependency Inversion Principle (DIP) – Depend upon abstractions. Do not depend upon concretions.
- The Interface Segregation Principle (ISP) – Many client specific interfaces are better than one general purpose interface.

In [BF07] (technical report) the authors propose Modifiability Tactics that are architectural design decisions. These decisions affect parameters based on coupling and cohesion.

Examples:

- Split a Responsibility – In order to reduce the cost of modifying a single responsibility.
- Abstract Common Services – In order to increase cohesion.
- Reduce coupling by use of encapsulation, use of wrapper, raise of abstraction level, use of intermediary, or restriction of communication paths.

Patterns are usually build on top of design principles. In [BF07] (technical report) the authors show how modifiability tactics are applied in architectural design patterns.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation to domain, however there might be automation-specific principles)
Relevance, Sustainability:	High (Software development should consider best practices)
Applicability:	Medium / High (Most principles come from industry, but validation is difficult)
Tool:	–
Preventive / Reactive / Analytical:	Preventive
Formalization:	Informal

Perspective:	Positive / Negative
Abstraction level:	Low / Medium / High
Benefit for sustainable systems:	Principles help to increase independence, reduce complexity, and improve modifiability.

References:

- [Oliver Ciupke. Problemidentifikation in objektorientierten Softwarestrukturen. PhD thesis, The University of Karlsruhe, 2002, page 133] [Ciu02], List of Design Heuristics (with references)
- [Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996] [Rie96] (book)
- [Nord R. Bachmann F., Bass L. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, 2007] [BF07] (technical report)
- Design Principles from Clean Code Philosophy [Mar09], http://www.objectmentor.com

4.1.2 Design Patterns

Goal: Provide solution for recurring design problems, incorporate variability and flexibility.

Short Description: A pattern describes a problem that occurs over and over again in software development, and describes the core of the solution to that problem. According to [GHJV95] (book) a pattern has four essential elements: 1) pattern name, 2) problem, 3) solution, and 4) consequences. [GHJV95] (book) divides design patterns into three categories: 1) creational patterns, 2) structural patterns, 3) behavioral patterns. In [Bus01] (book) *patterns for software architectures* are defined as "a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate".

There are several patterns that help with controlling dependencies and increasing independence of system parts. These patterns are likely to improve system maintenance and have a positive impact on sustainability.

Patterns that have influence on modifiability are for example:

- Facade – Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Bridge – Decouple an abstraction from its implementation so that the two can vary independently.
- Decorator – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The patterns in [GHJV95] (book) are primarily on object-oriented design level. But on architecture level one finds also collections of patterns.

In [Bus01] (book) the authors distinguish architectural patterns, design patterns, and idioms. An idiom is a low-level pattern specific to a programming language.

Examples for architectural patterns with influence on modifiability are:

- Layers – Helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
- Model-View-Controller – Divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface.
- Reflection – Provides a mechanism for changing structure and behaviour of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. An application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behaviour.

Each pattern introduces a certain degree of flexibility or variability. To determine which flexibility is necessary and to select the appropriate pattern is a challenge. Approaches in the context of product line engineering might help with eliciting variability requirements.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation to the domain, some patterns might be typical for usage in automation systems, e.g., regarding event-driven communication, real-time, etc.)
Relevance, Sustainability:	High (Patterns highly influence modifiability and therefore are highly relevant for achieving sustainability)
Applicability:	Medium (Patterns are applied in industry, but highly depends on developer knowledge and acceptance)
Tool:	–
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Patterns can be used to control flexibility and variability.

References:

- [Ralph Johnson, John Vlissides, Erich Gamma, Richard Helm. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995] [GHJV95] (book)
- [Frank Buschmann. A system of patterns. Wiley, repr. edition, 2001] [Bus01] (book)
- [Weihang Wu and Tim Kelly. Safety tactics for software architecture design. Proceedings of the 28th Annual International Computer Software and Applications Conference, 01:368-375, 2004] This article presents a method ("safety driven method") for software architecture design within the context of safety. The proposed method is centered upon extending the existing notion of architectural tactics to include safety as a consideration. For the selection of appropriate tactics (consistent with SEI tactics) the authors propose the construction of arguments for each candidate based on provided template.The method addresses the failures of components and failure behaviors through the interaction of components under consideration of safety requirements. [WK04] (conference paper)

4.1.3 Reference Architectures

Goal: Standardisation of architectures that are specific to an application domain.

Short Description: A reference architecture is an abstract software architecture, which defines structures and types of software elements, their allowed interactions and their responsibilities specific to an application domain. Structures are applicable for all systems within an application domain. They represent established basic constructions that contain collected experience of several engineering generations, and are supported by a large community of researchers and practitioners.

For example, in compiler construction it is common to split components into lexical analysis (Scanner), syntactical analysis (Parser), semantic analysis, and generators, [RH09] (book). Other examples are Quasar, [Sie06] (book), which is a reference architecture for business information systems and AUTOSAR, [AK] (webpage), providing a reference architecture for software in cars.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation to the domain, not automation specific)
Relevance, Sustainability:	High (See sustainability benefits below)
Applicability:	Medium (No experience reports found, no tool support)
Tool:	–
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive / Negative
Abstraction level:	Medium / High
Benefit for sustainable systems:	Reference architecture provide standardized solutions for domain-specific structuring. This also covers aspects of flexibility and modifiability, which influence sustainability.

References:

- [Ralf Reussner and Wilhelm Hasselbring. Handbuch der Software-Architektur. dpunkt.verlag, 2. edition, 2009, Chapter 17] [RH09] (book)
- Quasar: [Johannes Siedersleben. Moderne Softwarearchitektur. dpunkt.verlag, 2006] [Sie06] (book)
- Autosar: AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers, and tool developers. `http://www.autosar.org` [AK]

4.2 Reactive elimination of evolution problems

4.2.1 Approach: Evolution in the small

Goal: Refactoring of the architecture in order to better facilitate the extension by new functionality.

Short Description: This comprises the usage of small, fast and efficiently executable, systematic steps of adaptation. System is runnable the whole time and can stay in productive use. Evolution in the small is especially reasonable if it is embedded in a development method.

Some well-known refactoring techniques [FBB+99](book) focus on code level, but even on model level refactoring techniques are very promising. In Section 7.4 they describe an example refactoring approach on model level. In this example a new reporting component is introduced in a business application, the information flow to the component is initiated, after testing the components output the output is connected with the management component.

There are several criteria for the applicability of evolution on architecture level: 1) architecture is explicitly given, 2) architecture should have a reasonable abstraction level, 3) project must not be too large

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (No limitation of domain)
Relevance, Sustainability:	High (Improves actively sustainability)
Applicability:	High (Industrial applicable, tool support available)
Tool:	Refactoring tools exist, also provided by IDEs and their extensions.
Preventive / Reactive / Analytical:	Reactive
Formalization:	Formal
Perspective:	Not applicable
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Refactoring in the smalls helps keeping code clean, better understandable, having less duplication, better encapsulation, etc.

References:

- [Ralf Reussner and Wilhelm Hasselbring. Handbuch der Software-Architektur. dpunkt.verlag, 2. edition, 2009, Chapter 7] [RH09] (book)

4.2.2 Approach: Migration with DUBLO architectural pattern

Goal: Architecture pattern for smooth evolution of business information systems.

Short Description: The DUBLO pattern can be generalized and reused for other evolution scenarios. DUBLO is based on a partial duplication of business logic between legacy system and new middle tier. Although separation of concern principle is violated to some degree, one gains flexibility and the possibility of a smooth evolution.

Advantages are smooth evolution is possible, consistency of database, independence of database, and reusability. Disadvantages are new adapter between legacy and new business logic is necessary, functional access layer in legacy system (if not already present), possibly negative performance impact through additional layers, and risk of duplication of legacy code.

The pattern has been applied to migration of client-server information system of the KDO company in Germany.

Evaluation:

Development Phase:	Implementation, Maintenance
Relevance, Automation:	Low / Medium (Applied to client-server information system of KDO)
Relevance, Sustainability:	Medium (Helps with actively evolution of systems)
Applicability:	Medium (Industrial applicability shown, but no tool support)
Tool:	no tool support
Preventive / Reactive / Analytical:	Reactive
Formalization:	Formal
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Smooth evolution

References:

- [W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The dublo architecture pattern for smooth migration of business information systems: an experience report. In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, pages 117-126, 2004.] [HRJ+04] (conference paper)
- [Ralf Reussner and Wilhelm Hasselbring. Handbuch der Software-Architektur. dpunkt.verlag, 2. edition, 2009, Chapter 9] [RH09] (book)

4.3 Variability Strategies

This section surveys approaches that help to investigate the requirements of variability in software systems. Providing the appropriate degree of variability is important for sustainable software systems, since software systems should provide variability for parts that have to change during evolution. The kinds of sustainable systems, which we are addressing here, should in particular provide variability regarding the exchange and evolution of (underlying) technologies and therefore aim at separating business logic from technical layers. The presented approaches can be used to identify the technological dependencies and determine the demand for variation points.

4.3.1 Approach: Generative Programming (Czarnecki, Eisenecker)

Goal: Automatically create a highly customized and optimized intermediate or end-product.

Short Description: Generative Programming is a software engineering paradigm based on modelling software system families, such that, given a particular requirement specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

The approach proposes analysis and design methods and several implementation technologies. Analysis and design methods involve domain engineering, object-oriented analysis and design (OOA/D), and feature modelling.

Proposed implementation technologies comprise generic programming, template-based C++ programming, aspect-oriented programming, generators, static meta programming in C++, and intentional programming.

The principles of domain analysis and feature modelling are still relevant, especially in context of product line engineering, domain-specific languages, and model-driven development. The proposed technologies are outdated.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (Not limited to a domain)
Relevance, Sustainability:	Medium (Aims at increasing reuse, understandability and flexibility, which have influence on sustainability)
Applicability:	Low (Domain analysis and feature modelling relevant, but technology is outdated, successors can be seen in product line approaches and model-driven software development approaches.)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Informal
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Domain analysis and feature modelling can improve understanding and implementation of flexibility requirements, hence lead to an better evolution of system.

References:

- [Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming. Addison-Wesley, 2000] [CE00] (book)
- [Thomas Stahl and Markus Völter. Model-Driven Software Development Technology, Engineering, Management. Wiley, 2006, Section 4.4] [SV06] (book)

4.3.2 Summary: Product Lines

Product Lines represent a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way, (http://www.sei.cmu.edu/architecture/start/glossary/).

According to [CN02] (book) there are three essential activities for product line development: 1) Core Asset Development, 2) Product Development, and 3) Management.

In product line engineering variability considerations have a very high priority, in order to determine which parts should be manifested as core assets, which parts should be present in each product, and which are optional. Therefore product-line engineering incorporates a sustainability awareness.

References:

- [Jan Bosch. Design and Use of Software Architectures Adopting and evolving a product-line approach. Addison-Wesley, 2000] [Bos00] (book)
- [Jack Greenfield and Keith Short. Software Factories Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004] [GS04] (book)
- [Paul Clements and Linda Northrop. Software Product Lines Practices and Patterns. Addison Wesley, 2002] [CN02] (book)

4.3.3 Approach: Product Lines with purevariants

Goal: Pure::Variants is a variation modelling approach to model the variation found in a Software Product Family (SPF) in design and code artefacts as to enable large scale re-use.

Short Description: An important aspect of SPFs is managing the variability. A SPF offers a set of features spanning the scope of products it tries to cover. An individual product is (ideally) derived from a SPF by selecting a subset of the feature set applicable to the product in question. A variation model describes these available features, their constraints, and the relationship they have to the design and code artefacts.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium
Relevance, Sustainability:	Medium (Helps with variability management, establishing the right variability leads to sustainable systems)
Applicability:	Medium (Not applied at ABB BUs. However, two research projects in DECRC and SECRC have investigated the use of different variation modeling tools and both recommended the use of the Pure::Variants.)
Tool:	http://www.pure-systems.com/
Preventive / Reactive / Analytical:	Preventive
Formalization:	Informal
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Managing of variability helps to keep the critical parts, which are likely to change, flexible and modifiable.

References:

- Pure::Variants website: http://www.pure-systems.com/

4.3.4 Approach: COSVAM: COVAMOF Software Variability Assessment Method

Goal: The goal is to determine whether, when, and how variability should evolve in a Software Product Family (SPF).

Short Description: COSVAM is the COVAMOF (ConIPF Variability Modelling Framework) Software Variability Assessment Method, a structured approach for assessing the variability provided and required by a SPF. The approach does this both for time, i.e., when is the variability required and provided, and in space, i.e., what functionality needs to be variable. The found variability mismatch provides viable information for release planning and planning the evolution of the SPF.

COVAMOF is supported by a tool-suite, called COVAMOF-VS. This tool-suite is implemented as a combination of Add-Ins for Microsoft Visual Studio and provides an integrated variability view on the active project, which contains the product family artefacts.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium
Relevance, Sustainability:	Medium (Helps with variability management, establishing the right variability leads to sustainable systems)
Applicability:	Medium (Industrial validation presented in journal paper, tool support is focused on Microsoft Technology)
Tool:	COVAMOF-VS (Link not found)
Preventive / Reactive / Analytical:	Preventive / Analytical
Formalization:	Informal
Perspective:	Not applicable
Abstraction level:	Medium / High
Benefit for sustainable systems:	Variability modelling is important for sustainability because is helps with keeping the critical parts, which are likely to change, flexible and modifiable.

References:

- [S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch. Cosvam: a technique for assessing software variability in software product families. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, pages 458-462, 2004] [DSNB04] (conference paper)
- [Marco Sinnema and Sybren Deelstra. Industrial validation of covamof. Journal of Systems and Software, 81(4):584-600, 2008] [SD08] (journal paper)
- [Sybren Deelstra, Marco Sinnema, and Jan Bosch. Variability assessment in software product families. Inf. Softw. Technol., 51(1):195-218, 2009] [DSB09] (journal paper)

4.4 Automating Software Development

In this section we survey approaches that aim at automating software development and increase the abstraction level of the implementation. For sustainable software systems the application of model-driven development increases the abstraction level, which is suitable to hide technical details. For example, the architecture and business logic of an application are described at an technology independent level and only transformations, which are the drivers of model-driven techniques, carry technology specific information. If such systems evolve, ideally only the transformations need to be adapted.

4.4.1 Approach: Model Driven Architecture (OMG)

Goal: Model Driven Architecture (MDA) aims at strictly separating domain-specific, technical and implementation specific aspects of a software system in order to establish portability and interoperability of software.

Short Description: The MDA is a standardization initiative of the OMG for model-driven software development. The core building blocks are:

- UML 2.0 - The unified modelling language comprising 14 diagrams for modelling different views of object-oriented software systems.
- Meta Object Facility (MOF) - Meta Object Facility describes a meta meta model which is the basis for UML2.0 and MDA-conforming tools
- XML Metadata Interchange (XMI) - A mapping from MOF to XML
- Three kinds of models (PIM, PSM, PDM) - Platform-independent model (PIM) for describing the business logic. A platform-specific model (PSM) is created from a PIM via model transformation and covers platform-specific properties (e.g. for J2EE, .NET, or other implementation platforms). The platform description model (PDM) is the meta model of the target platform.
- Multi-Stage Transformation - Source code is obtained via several subsequent model-to-model transformations. QVT (Query/View/Transformation), in the model-driven architecture, is a standard for model transformation defined by the Object Management Group.
- Action Languages - for modelling of procedural behaviour.
- UML profiles, Executable UML (see Section 4.4.2)

The MDA has coined many established terms in the standardization process. It serves as a reference model for the whole MDSD community.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation of domain)
Relevance, Sustainability:	Medium (Generation of code helps with avoiding coding errors)
Applicability:	Medium (A survey of experience reports in [MC07] (journal article) shows that MDA has been applied in industry several times in various domains, but maturity has still some limitations.)
Tool:	A list of tools is available at http://www.omg.org/mda/committed-products.htm
Preventive / Reactive / Analytical:	Preventive
Formalization:	Informal
Perspective:	positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Rise abstraction level closer to domain, generate repetitive code.

References:

- [OMG. MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003] [OMG03] (standard specification)
- Official Website of OMG to MDA, www.omg.org/mda, Specifications, UML Profiles, etc.
- [Thomas Stahl and Markus Völter. Model-Driven Software Development Technology, Engineering, Management. Wiley, 2006, Chapter 12: The MDA Standard] [SV06] (book)

4.4.2 Approach: xtUML - Executable UML

Goal: Build extensive UML specifications that can be executed and simulated without generating code for more cost effective development.

Short Description: xtUML is a profile for UML2.0 that allows enhancing UML models with an action language to be executable. The models can be simulated with additional simulators, and platform-specific code can be generated in different programming languages. The main benefits are that a specification can be evaluated through simulation during early development stages, and that different hardware platforms or programming languages can be supported with relative ease.

SAAB has used the approach to simulate and generate code for a military tactical simulator application. They generated ADA and C++ code from 70KLOC of action language code. In addition to the 450KLOC generated code they added 120KLOC of manually implemented code. The whole approach is in line with the MDA initiative by the OMG.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation to domain))
Relevance, Sustainability:	Medium (Higher abstraction level, hiding of low-level details improves understanding, avoids consistency problems between artefacts, separation of domain-logic and technology)
Applicability:	High (Has been applied in industry, e.g., at SAAB)
Tool:	–
Preventive / Reactive / Analytical:	Preventive
Formalization:	Informal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Higher abstraction level, better hiding of low-level details.

References:

- [Stephen J. Mellor and Marc J. Balcer. "Executable UML: A Foundation for Model-Driven Architecture". Addison-Wesley, 2002] [MB02] (book)
- [Erik Wedin (SAAB). "Model-Driven Architecture and xtUML in Practice" IESF Conference 2009, Seattle, USA] [EW09] (conference paper)
- http://en.wikipedia.org/wiki/Executable_UML

4.4.3 Approach: Architecture-Centric MDSD

Goal: The goal of MDSD is to integrate automation of infrastructure code generatio, and the minimization of redundant infrastructure code in application development with a special emphasis on software architecture.

Short Description: Architecture-Centric MDSD is a specialization of MDSD that conceptually overlaps with MDA. In contrast to the primary goals of the OMG for MDA, interoperability and software portability, AC-MDSD aims at increasing development efficiency, software quality, and reusability. Software developers are relieved from tedious and error-prone routine work by generation of infrastructure code, which mostly serves to establish technical coupling between infrastructure and application that facilitates the development of domain-specific code on top of it, e.g. J2EE/EJB.

The properties of AC-MDSD are:

- Architecture-centric design – No use of platform-specific models (like in MDA), but platform-independent models only. The maintenance effort for intermediate results is reduced and consistency problems are avoided.
- Forward engineering – No round-trip engineering. Thus no reverse engineering of models. Changes are done only in the model (design). Thus model is always consistent with generated source code.
- Model-to-Model transformation for modularization only – Intermediate models are considered as implementation details, that are invisible to developers.
- Source code generation without explicit use of the target metamodel – There is no meta model representing source code, but generation is based on generator templates. A reference architecture implementation which demonstrates the realisation in source code is used to derive generator templates. Quality of generated code is controlled by quality of template code.

AC-MDSD serves as foundation for several modern tools and frameworks, e.g. Eclipse Modeling Tools, Section 4.4.4.

Sustainable systems benefit from AC-MDSD if the target software system involves a standard platform which is ideally shared among multiple software projects (reuse potential). Nevertheless, as all model-driven techniques, AC-MDSD has strong tool dependencies, therefore, the development cycles and the maturity of tools impacts the sustainability. The application of MDSD implies high learning efforts and a steep learning curve.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (Not limited to domain)
Relevance, Sustainability:	Medium (See benefits below)
Applicability:	High (tool support available)
Tool:	An implementation of the approach is available, for example, by Eclipse Based Modeling (Section 4.4.4), but it is not restricted to Eclipse.
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Generation of infrastructure code, separation of domain-logic and technology.

References:

- [Thomas Stahl and Markus Völter. Model-Driven Software Development Technology, Engineering, Management. Wiley, 2006, Chapter 11, 18, 19, 20] [SV06] (book)
- [Ralf Reussner and Wilhelm Hasselbring. Handbuch der Software-Architektur. dpunkt.verlag, 2. edition, 2009, Chapter 5: Model-driven Software Development] [RH09] (book)

4.4.4 Summary: Eclipse-Based Modelling

Eclipse modelling projects provide implementations and tooling for the Model-Driven Architecture and Model-Driven Software Development approaches mentioned in previous sections. Eclipse-based tooling represents a de-facto standard for model-driven techniques in practice. The following projects are the most relevant to MDSD.

Abstract Syntax Development

- **Eclipse Modeling Framework (EMF)**: a modelling framework and code generation facility for building tools and other applications based on a structured data model.

Concrete Syntax Development

- **Graphical Modeling Framework (GMF):** provides a generative component and runtime infrastructure for developing graphical editors based on EMF and Graphical Editing Framework (GEF).
- **Textual Modeling Framework (TMF):** provides tools and frameworks for developing textual syntaxes and corresponding editors based on EMF.

Model Transformation

- **Model to Model Transformation (M2M):** an extensible framework for model-to-model transformation languages, with an exemplary implementation of the QVT Core language.
- **Model to Text Transformation (M2T):** focuses on technologies for transforming models into text (typically language source code and the resources it consumes)

References:

- **Eclipse Modeling Project**
  - [Erich Gamma, Lee Nackmann, and John Wiegand, "eclipse Modeling Project A Domain-Specific Language (DSL) Toolkit", Addison-Wesley, 2009] [GNW09] (book)
  - Website: http://www.eclipse.org/modeling
- **Open Architecture Ware (OAW)**
  - The components of OAW (Workflow Engine, Xtext, Xpand, Xtend, Check) are now part of the Eclipse Modeling Project
  - Website: http://www.oaw-dev.de

4.4.5 Approach: SQL Server Modeling CTP (old name: OSLO)

Goal: The goal of SQL Server Modeling CTP is to build applications out of data and meta data stored in database.

Short Description: The SQL Server Modeling CTP is a set of tools for building applications out of data. This means the applications are completely described in data and metadata that is contained within a database. As data, the application definition can be viewed and edited in a variety of forms. The SQL Server Modeling CTP is a Microsoft Technology and formerly known as OSLO.

The components of the SQL Server Modeling CTP are:

- "M" is a textual language for defining schemas, queries, values, functions and domain-specific languages for SQL Server databases
- "Quadrant" is a customizable tool for interacting with large datasets stored in SQL Server databases
- SQL Server Modeling Services (formerly the "Oslo" Repository) is a SQL Server role for the secure sharing of models between applications and systems

Sustainable systems benefit from SQL Server Modeling CTP since its application enables changing of data representation in central places. Moreover an integrated meta-data handling is possible, i.e., the representation of data in database and in implementation is synchronized by design. The approach is proprietary therefore there is a risk of vendor lock-in.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (No limitation to domain)
Relevance, Sustainability:	Medium
Applicability:	Low (Few practical evidence)
Tool:	Integrated in Visual Studio, http://msdn.microsoft.com/en-us/library/cc709420.aspx
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Increase abstraction level, Hide low-level details, generation of infrastructure code for technology stack dependencies.

References:

- Official page at MSDN: http://msdn.microsoft.com/en-us/library/cc709420.aspx

4.4.6 Approach: Constructor MDRAD

Goal: Apply approach for easier access to database by generation of object-representation of database schema.

Short Description: Constructor/MDRAD is a toolkit integrated to the Visual Studio 2002, 2003, 2005 or 2008 IDE. It allows developers to build a set of classes that closely follow a database schema, and provides an extensive API with CRUD (Create, Read, Update, Delete) functionality at run time, without the need to write or hard-code any SQL statements.

The Object Run Time assemblies handle data inheritance, one-to-one, one-to-many and many-to-many relationships. Data is automatically lazy-filled when relationship paths are navigated and object queries provide a way to retrieve data that follow a specific criteria.

By applying this approach the evolution of database schema becomes easier, since after changes to database schema the access code could be re-generated and does not need to be adapted manually.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium
Relevance, Sustainability:	Medium
Applicability:	Low (Few practical evidence)
Tool:	(commercial license, academic license) http://www.i3design.co.uk/constructor/mdrad/
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Generation of code for handling database content

References:

- Official Website: http://www.i3design.co.uk/constructor/mdrad/

4.4.7 Approach: Stratego XT

Goal: In order to rise the abstraction level of the code closer to domain terminology use domain-specific languages and generate fine-grained code by applying program transformation and code generation. By applying this approach evolution of business logic can be kept more independent from evolution of technological platform.

Short Description: Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction.

The XT toolset offers a collection of extensible, reusable transformation tools, such as parser and pretty-printer generators and grammar engineering tools. Stratego/XT supports the development of program transformation infrastructure, domain-specific languages, compilers, program generators, and a wide range of meta-programming tasks.

The toolset is implemented using C, but for some parts Java versions are available as well. Closely related is the Spoofax Language Workbench, which is a platform for developing textual domain-specific languages with Eclipse editor plugins.

The approach seems to be more on a scientific level, without much practical experience.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium
Relevance, Sustainability:	Medium
Applicability:	Low (Few practical evidence)
Tool:	http://strategoxt.org/
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal
Perspective:	Positive
Abstraction level:	Medium / High
Benefit for sustainable systems:	Increase abstraction level, Hide low-level details, Generation of infrastructure code for technology stack dependencies.

References:

- Official Website: http://strategoxt.org/
- Spoofax Language Workbench, http://strategoxt.org/Spoofax/WebHome

4.5 Development Process Decisions

Topics reviewed in this section have an introductory purpose, their description in intentionally kept high-level without detailed overview of exact approaches. The only exclusion are sections dedicated to the agile methods and consistency checking, which intentionally provide some approaches in order to demonstrate the peculiarity of issues described in the corresponding subsections.

The following topics are reviewed:

- Agile methods
- Knowledge transfer, documentation, UML
- Consistency between artefacts
- Quality assurance strategies
- Team organisation strategies
- Development environment Strategies

4.5.1 Agile methods

As agile methods become more and more popular in the broader software development community, they were included into the literature research [Bab09, SA08, SD07]. Due to the emerging character of agile methods, their applicability for sustainable systems is highlighted in the following. In the conclusion, their relevance and applicability are critically reflected.

The advantages of agile methods, like flexibility and quick reaction to changes, could be interesting for the automation domain with its typically heavy weight processes. However, it is unclear to what extend such methods can be adopted for projects which develop sustainable systems. For safety-critical systems of the automation domain, heavy weight approaches with comprehensive documentation and extensive planning are necessary to comply with safety regulations. Therefore, before adapting an agile development process, one has to consider and analyse its impact on different project stages (system's life phases).

This section provides an overview of aspects of agile methods and introduces the following topics:

- Properties and introduction of agile methods into the organization process (brief method overview, method selection and traps), 4.5.1.1
- Long-term impact of agile methods on software life cycle (expert opinion), 4.5.1.2
- Architecture modelling and agile methods, 4.5.1.3
- An example of an approach assisting in the selection of an agile method corresponding to the organisation's needs, 4.5.1.4
- An example of a hybrid agile method with explicitly considers architecture and design (for elimination of maintenance and evolution issues common for poor agile methods), 4.5.1.3

4.5.1.1 Properties and introduction of agile methods into the organization process

This subsection provides a brief overview of the most known agile methods, methods selection and several issues connected to the introduction of agile methods into the processes of an organization.

First of all, one has to ensure, that the right method for the organization's needs is selected. Considering the available variety of agile methods, this is a non-trivial task. There are many agile methods, of which the most known are:

- Extreme Programming (XP), [K. Beck and C. Andres, Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004] [BA04] (book). It is one of the most famous agile methods, judging on the amount of publications. Being strongly dogmatic and quite high-level, it is currently rather rarely used in it's original form. The survey [Ver09] shows that the Scrum and XP hybrid method is more likely to be fit for many projects, compared to pure XP. However, many of the XP practices were overtaken by other agile methods.
- Scrum, [K. Schwaber and M. Beedle, Agile Software Development with Scrum. Pearson Studium, 2008] [SB08] (book). According to [Ver09], Scrum is the most widely spread agile method nowadays. It's advantages are flexibility and simplicity. It is based on team spirit and forwards self-motivation.
- Crystal (clear) methods, [A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Longman, Amsterdam, 2004] [Coc04] (book). Is a method family consisting of 7 members, each destined for a certain type of projects. The method family is more dogmatic than XP and often more flexible, depending on exact project needs. The methods are, however, still under development (only brief descriptions are

available). The most comprehensively described is the "clear" method variant (for small teams with not more than 6 persons).

- Feature Driven Development (FDD), [S. R. Palmer, M. Felsing, and S. Palmer, A Practical Guide to Feature-Driven Development. Prentice Hall International, 2002] [PFP02] (book). This method is based on meta modelling and a set of best practices approved in industry. Although, there is are a dedicated community (http://www.featuredrivendevelopment.com/) and various certifications available, only a few publications and supporting software concerning FDD are currently available.

An example of the approach that may assist in selecting the right method for the organisation's needs in described in 4.5.1.4. One has to consider that the introduction of a new development method is usually connected to the restructuring of already existing organisational processes, and therefore creates additional overhead and costs.

Furthermore, agile methods are usually applied for smaller projects. In order to support large scale development, special techniques have to be adapted together with the agile methods. [CH01] mentions an agile project with ca. 250 members and, according to the materials provided by http://scrumcenter.net/, Scrum (one of the agile methods) is known to be successfully adapted for 500+ team members.

As systems in the automation domain are highly complex, basic principles of agile methods, such as "code is documentation" and "avoidance of waste" in form of architectural modelling, likely do not fit to the needs of software projects in the automation domain. Experts believe that poor agile methods have a rather negative impact on system maintenance and evolution 4.5.1.2. Hybrid agile methods extended with additional practices (like architectural modelling) are therefore intended to prevent issues during system maintenance and evolution, such as architectural erosion and low system understandability that are common troubles of long living systems. More on the reconciliation of agile methods and architecture modelling, as well as a set of related references can be found in 4.5.1.3. An example of a hybrid approach is presented in 4.5.1.5.

References:

- [G. Goth. Agile Tool Market Growing with the Philosophy. Software, IEEE, 26 Issue:2, 88 – 91, 2009] [Got09] (journal paper)
- [M. Taromirad, R. Ramsin, "An Appraisal of Existing Evaluation Frameworks for Agile Methodologies", In Proceedings of the 15th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'08), Northern Ireland, 2008, pp. 418 – 427] [TR08] (conference paper)
- [P. Lappo, H. C.T. Andrew, "Assessing Agility", Lecture Notes on Computer Science, Springer-Verlag, Germany, 2004, pp. 331 – 338] [LA04] (journal paper)
- [D. Turk, R. France, B. Rumpe, "Assumptions Underlying Agile Software Development Processes", Journal of Database Management, Idea Group Inc., October-December 2005, pp. 62 – 87] [TFR05] (journal paper)
- [P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, Agile Software Development Methods: Review and Analysis, VTT Publication, Finland, 2002] [ASRW02] (conference paper)
- [P. Abrahamsson, J. Warsta, M. Siponen, J. Ronkainen, "New Directions on Agile Methods: A Comparative Analysis", In Proc. of the 25th International Conference on Software Engineering (ICSE'03), Oregon, 2003, pp. 244 – 254] [AJaJR03] (conference paper)
- [J. Koskela, Software Configuration Management in Agile Methods, VTT Publication, Finland, 2003] [Kos03] (conference paper)

- [L. Williams, W. Kerbs, L. Layman, A. Anton, "Toward a Framework for Evaluating Extreme Programming", In Proc. of the 8th International Conference on Empirical Assessment in Software Engineering (EASE 04), Edinburgh, 2004, pp. 11 – 20] [WKLA04] (conference paper)

- [E. Germain, P. Robillard, "Engineering-based Processes and Agile Methodologies for Software Development: a Comparative Case Study", The Journal of Systems and Software, Elsevier, February 2005, pp. 17 – 27] [GR05] (journal paper)

- [A. Qumer, B. Hendersson-Sellers, "Comparative Evaluation of XP and Scrum Using the 4D Analytical Tool (4-DAT)", In Proceedings of the European and Mediterranean Conference on Information Systems (EMCIS), Spain, 2006] [QHS06] (conference paper)

- [M. Pikkarainen, U. Passoja, "An Approach for Assessing Suitability of Agile Solutions: A Case Study", In Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005), UK, June 2005, pp. 171 – 179] [PP05] (conference paper)

- [D. Turk, R. France, B. Rumpe, "Limitations of Agile Software Processes", In Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2002), Italy, May 2002, pp. 43 – 46] [TFR02] (conference paper)

- [K. Beck et al., "Manifesto for Agile Software Development", Available at http://www.agilemanifesto.org] (on-line article)

- [Agile Alliance, "Agile Principles", Available at http://agilealliance.org] (on-line article)

- [K. Conboy, B. Fitzgerald, "Toward a Conceptual Framework of Agile Methods: A Study of Agility in Different Disciplines", In Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research, CA, USA, 2004, pp. 37 – 44] [CF04] (conference paper)

- [Mohan K., Ramesh B., Sugumaran V., Integrating Software Product Line Engineering and Agile Development. Software, IEEE, 27 Issue:3, 48 – 55, 2010] [MRS10] (journal paper)

- [Laanti, Nokia Corporation, 2008, Paper, Implementing Program Model with Agile Principles in a Large Software Development Organization] [Laa08]

- [Koehnemann, 2009, Paper, Experiences Applying Agile Practices to Large Systems] [Koe09]

- [I. Christou, S. Ponis, and E. Palaiologou. Using the Agile Unified Process in Banking. Software, IEEE, 27 Issue:3, 72 – 79, 2010] The Agile Unified Process (AUP)-a hybrid approach designed by Scott Ambler combining RUP with agile methods to a successful project in the banking sector. The project achieved on-time delivery within budget, integrating heavy legacy back-end application systems with newly reengineered client user-interface applications on a modern service-oriented architecture (SOA) platform. [CPP10] (journal paper)

- [Goetzenauer. Agile Methoden in der Softwareentwicklung: Vergleich und Evaluierung. Master's thesis, 2005] [Göt05] (diploma thesis, in German)

- [M. A. Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. Joint Working IEEE/IFIP Conference on Software Architecture, 2009 and European Conference on Software Architecture. WICSA/ECSA 2009., pages 81 – 90, 2009] [Bab09] (conference paper)

- [O. Salo, P. Abrahamsson. Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum. Software, IET, 2, Issue 1, pages 58 – 64, 2008] [SA08] (journal paper)
- [K. Silva and C. Doss. The growth of an agile coach community at a fortune 200 company. AGILE 2007, pages 225 – 228, 2007] [SD07] (conference paper)
- [K. Beck and C. Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004] [BA04] (book)
- [Versionone. State of agile survey, 2009] [Ver09] (survey)
- [K. Schwaber and M. Beedle. Agile Software Development with Scrum. Pearson Studium, 2008] [SB08] (book)
- [A. Cockburn. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Longman, Amsterdam, 2004] [Coc04] (book)
- [S. R. Palmer, M. Felsing, and S. Palmer, A Practical Guide to Feature-Driven Development. Prentice Hall International, 2002] [PFP02] (book)
- [A. Cockburn and J. Highsmith. Agile software development: The people factor. Computer, pages 131 – 133, 2001] [CH01] (journal paper)
- Feature Driven Development (FDD), http://www.featuredrivendevelopment.com/ (dedicated community)
- Scrum Center, http://scrumcenter.net/

4.5.1.2 Maintenance and Agile Development, Long-term Life Cycle Impact of Agile Methodologies

This subsection introduces research on the impact of agile methods on the maintenance (which are as well related to the evolution) of systems. Certain aspects, nevertheless, like lack of documentation and bad architecture, are also relevant for other methodologies.

Maintenance of system developed using agile methods, pros and cons ([KMSN+06]:

- (negative) Current agile software methodologies only represent a few implementation details of the product development process, and because of this "low-level" approach they may build software that meet short-term individual project needs, but that do not necessarily lead to software systems suitable for longterm "enterprise" software (Heydt M.).
- (negative) "Compare the savings made through agile development to the costs of maintaining a product for the next 20 years. Agile development is only postponing costs which then come later in the maintenance phase" (Sneed H.).
- (negative) A lack of or insufficient system documentation may lead to increased system complexity, deteriorated maintainability, lack of system familiarity, difficulties to assess the impact of change and side effects, and confusion in the already difficult and complex maintenance task (Kajko-Mattsson M.).
- (neutral) The small role that software architecture plays in agile methodologies such as XP and Scrum is an impediment to long-term software maintenance, eventually makes a system unmaintainable. Emerging agile methodologies such as Feature-Driven Development (FDD) and the more structured versions of Crystal recognize the role and importance of software architecture, even if at a high level (Lewis G.).

- (positive) Software maintainability coupled with agile development methodologies can only be effective if built upon other best practices. Top-of-mind best practices include: 1) Team Building 2) Architecture 3) Test-driven development practices, architecture review and code inspection 4) Project Management. Agile enhancements include increased visibility, task and resource tracking (Siracusa D.).

Most agile methods assume that development starts from scratch and ends with a release — postrelease maintenance is not covered. There is no guarantee that the code can serve as documentation ("the code is the documentation"), if the system was originally developed using different methods. This has a rather negative impact on the system evolution, as typical troubles of long living systems, a) architecture erosion and b) system's understandability, are not concerned during the agile development.

References:

- [Kajko-Mattsson et al.. Long-term Life Cycle Impact of Agile Methodologies. 22nd IEEE International Conference on Software Maintenance, 2006. ICSM '06. Pages 422 – 425, 2006] Discussion of the long-term life cycle impact of agile methodologies. [KMSN+06] (conference paper)

- [Hanssen G.K., Yamashita A.F., Conradi R., Moonen L., Maintenance and agile development: Challenges, opportunities and future directions. IEEE International Conference on Software Maintenance, 2009. ICSM 2009. Pages 487 – 490, 200] Investigation of software entropy issue through an industrial case study, overview of the literature on this topic (focus on the detection of code smells and their treatment by refactoring). Conclusion: in order to remain agile despite of software entropy, developers need better support for understanding, planning and testing the impact of changes. However, it is exactly work on refactoring decision support and task complexity analysis that is lacking in literature. [HYCM09] (conference paper)

4.5.1.3 Architecture modelling and agile methods

Most of the agile methods consider architectural modelling to be superfluous. Most of the agile methods offer only high-level practices on explicit system design and architecture modelling. Therefore, multiple hybrid approaches were proposed to overcome this issue. Questions concerning the role and potential insertion of architecture modelling in agile methods and their reconciliation are becoming matters of high interest not only in research, but also in practice.

Explicit design and architecture modelling have the following benefits [CBBG02, Gar00]:

- Communication: architecture may be used as a focus of discussion by system stakeholders;
- Understanding: presenting a complex system at an easier abstraction level;
- Analysis: consistency checking, conformance to constraints and quality attributes, dependence analysis;
- Reuse: architectural descriptions support reuse at multiple levels and across a range of systems (styles, patterns, components, frameworks, code); existing components can be considered during design (COTS, in-house components, commissioned or off-shore);
- Management: evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies and potential risks (cost-estimation, mile stone organization, dependency analysis, change analysis, staffing);
- Implementation support: provides a partial blueprint for development by indicating the major components and dependencies between them;

- Evolution: exposure of the dimensions along which a system is expected to evolve

From the perspective of sustainable software architectures, only agile methods with explicit architecture considerations are recommended. Unfortunately, most of the work in this area is high-level and only discusses advantages of the reconciliation of agile methods and explicit architecture and design (you can find a set of reference below). For an example of a practical hybrid agile method with explicitly considered architecture and design refer to 4.5.1.3.

References:

- [P. Abrahamsson, M. A. Babar and P. Kruchten. Agility and Architecture: Can They Coexist? IEEE Software, 27 , Issue:2, 16 – 22, 2010] A good introduction into the problem. This article discusses the roots of misunderstanding architecture modelling and agile methods contradictions. It questions the general challenge of their reconciliation and gives several general advices, but provides no practical method. [AABK10] (journal paper)
- [P. Kruchten. Voyage in the Agile Memeplex. ACM 1542-7730/07/0700, Volume 5, Issue 5:1, 2007] Points out several problems in the agile community: community-specific terminology; usual absence of context (where to apply the method, where the method was useful); absence of established understanding of certain process parts and definition of waterfall method; attempts to create a culture out of agile methods. [Kru07] (journal paper)
- [S. Ambler. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. Wiley, 2002] Provides an introduction into agile methods, UML-modelling and describes general agile modelling methodologies. However all advices are kept very general, mostly concentration on the "best practices" of Extreme Programming (XP, one of the agile methods) and are more concerning a very general management level. [Amb02] (book)
- [M. Isham. Agie Architecture is possible - You first have to believe! Conference Agile, 2008. AGILE '08, pages 484 – 489, 2008] Presents a project success report where architecture and Scrum (one of the agile methods) were joined. This report however neither gives any details on architecture modelling (if it was used and how), nor proposes any method or process for their reconciliation. [Ish08] (journal paper)
- [D. Falessi, G. Cantone, S. Alessandro Sarcia, G. Calvaro, P. Subiaco, and C. D'Amore. Peaceful Coexistence: Agile developer perspectives on Software architecture. Software, IEEE, 27, Issue:2, 23 – 25, 2010] An exploratory study conducted by IBM and University of Rome Tor Vergata. It describes an agile developer perspective on the architecture. It shows that in 95% of cases, developers consider a focus on architecture as important, but it does not state what is understood by architecture in this case. It does not propose any method. [FCS+10] (journal paper)
- [B. Nuseibeh. Weaving Together Requirements and Architectures. Computer, 34, Issue:3, 115 – 119, 2001] The proposed method is an adoption of the spiral life-cycle model and the author describes it as a complementary to XP (Extreme Programming, one of the agile methods). However, the article is on a very high level, it gives no exact details about the method itself. [Nus01] (journal paper)
- [P. Clements, F. Bachmann, L. Bass, D. Garlan. Documenting software architectures: Views and beyond (SEI series in software engineering), 2002] [CBBG02] (book)
- [David Garlan. Software architecture: a roadmap. In Proceedings of the Conference on The Future of Software Engineering, pages 91 – 101, 2000] [Gar00] (conference paper)

4.5.1.4 Approach: CEFAM Comprehensive Evaluation Framework for Agile Methodologies (Taromirad)

Goal: Assist in the selection of an agile method corresponding to the organisation's needs.

Short description: This approach supports the selection of an appropriate agile methodology. The proposed Comprehensive Evaluation Framework for Agile Methodologies (CEFAM) [TR08] (workshop paper) is an evaluation tool for project managers and method engineers. It is based on a hierarchical (and mostly quantitative) evaluation criterion set, presented in Figure 4.6.

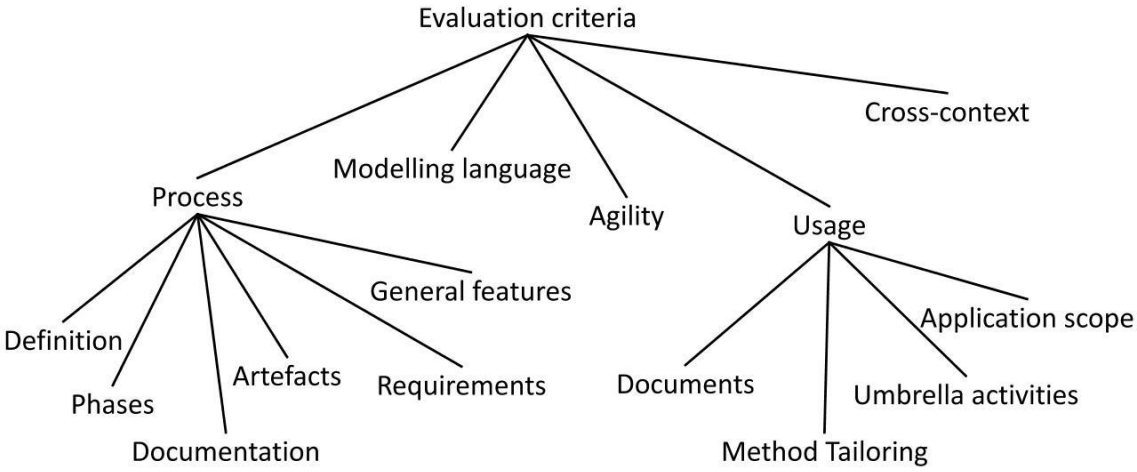

Figure 4.6: CEFAM Evaluation Criteria (structure)

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (Not automation specific, however, no domain limitation)
Relevance, Sustainability:	Medium (The decision to select a particular agile method influences system properties)
Applicability:	Low / Medium (Industrial validation is unclear, however, this framework is built on top of already existing methods eliminating their drawbacks)
Tool:	–
Preventive / Reactive / Analytical:	Preventive / Analytical
Formalization:	Informal
Perspective:	Positive
Abstraction level:	High
Benefit for sustainable systems:	Goal oriented selection of a method that suits the needs of an organisation

References:

- [Taromirad M. and Ramsin R., CEFAM: Comprehensive Evaluation Framework for Agile Methodologies. 32nd Annual IEEE Software Engineering Workshop, pages 195 – 204, 2008] [TR08] (workshop paper)

4.5.1.5 Approach: Agile Architecture Interactions (Madison)

Goal: Reconcile agile methods with architecture modelling.

Short description: This is a practice-oriented approach: A combination of agile methods and architectural modelling which uses agile techniques to derive good architectures. It is presented in a form of a framework based on Scrum (agile method), XP (agile method) and sequential project management. It proposes to use an architecture for communication, quality attributes and technology stack establishment purposes. It requires an architect who plays a central role and uses design patterns to define further forms of implementation. The proposed approach does not concern requirements engineering and seems to target organizations having only one software system type and thus similar project types (in the article: insurance software systems) as it is should use former architectural decisions. However, this assumption may hold for the automation domain.

Evaluation:

Development Phase:	Design / Implementation
Relevance, Automation:	Medium (Not automation specific, domain seems to be limited. However, this assumption may to hold for the automation domain)
Relevance, Sustainability:	Medium (Introduces architectural modelling into agile methods, therefore, cares about elimination of troubles during later system's evolution phase)
Applicability:	Medium (Not enough information for a complete adoption of proposed method, however, has been successfully applied in industry throughout multiple projects)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Informal
Perspective:	Not applicable
Abstraction level:	High
Benefit for sustainable systems:	Supports architectural modelling and, therefore, improves the impact on maintenance

References:

- [J. Madison. Agile Architecture Interactions. Software, IEEE, 27 , Issue: 2, 41 – 48, 2010] [Mad10] (journal paper)

4.5.2 Knowledge Transfer, Documentation, UML

The main goal of knowledge transfer is to prevent the loss of knowledge about the system and used technologies (highly relevant for evolution and maintenance of the long living systems) [BSB08] (book):

- General approaches for the knowledge transfer [BSB08] (book). Sharing knowledge between team members (or multiple teams) help to avoid situations, where suddenly nobody is able to maintain some system parts or carries knowledge about certain technologies. In-team rotation, regular meetings and reviews enable team members to take care of some system parts and processes even in case the main specialist responsible for it out of some reason is unavailable. Approaches that are introduced in the following sections are, however, only complimentary to documenting systems and are more helpful during implementation or bug fixing phases. Knowledge transfer approaches are presented in the Subsection 4.5.2.1.
- Documentation artefacts. Documentation artefact are text documents (specifications, manuals, requirements specifications and etc.) and models. Such artefacts are useful for performing activities during implementation and maintenance phases. Team members (specialists) may use them in order to obtain information about the system or system changes (design and/or implementation changes) required to perform their activities. Such artefacts, if being regularly updated, help to reduce reduces overhead connected to the system understanding. For long living systems in automation domain sufficient system documentation is highly relevant, as such systems are very complex and initial team members are often unavailable for support. Subsection 4.5.2.2 provides several references discussing positive influence of documentation on system maintenance activities.

4.5.2.1 General Approaches for Knowledge Transfer

General knowledge transfer approaches are used along with documentation to ensure knowledge transfer in the team. These are common practices widely used in the IT community. Certain knowledge sometimes is carried only by single person, thus making the whole team highly dependant on it. Knowledge transfer approaches help to avoid having the only available technology or system "experts" in the team . They also help to maintain overall knowledge state inside the team and support learning from the past. However, these approaches seem to be more useful during system implementation and bug fixing, as they tend to effectively support equal level of technical knowledge in the team during some shorter periods of time, but do not scale sufficiently enough over a longer period of time (long living is a common system attribute in the automation domain). Main general knowledge transfer approaches are ([BSB08]):

- Knowledge inventory table
- Retrospective meetings
- Knowledge database
- Review
- Meetings / Groups
- In-Team rotation

As these approaches do not prevent loss of knowledge about system and it's modifications during a longer period of time, they have to be seen as complimentary to documentation.

Knowledge inventory table helps to discover knowledge and abilities distribution in teams. Long living system often tend to get obsolete parts, e.g. implemented in an old/legacy programming language. Specialist able to work with these parts are likely to disappear (e.g. leave the company) over time. Therefore, it is important to assure that the knowledge they have is shared with newcomers and other team members. Knowledge inventory table may help keeping an overview of such knowledge in the team. It structurally shows which person carries which knowledge and abilities, therefore making gaps in the knowledge distribution visible.

An example of a knowledge inventory table:

–	**Importance**	**Main knowledge representer**	**Depth of knowledge**	**Deputy**	**Note**
System					
Core system	1	Smith	2	Jones	–
Automatic functionality	3	Brown	2	–	–
GUI functionality	2	Taylor	2	–	Taylor is leaving the company in 3 months
Technology					
.Net	1	Johnson	3	Jones	Johnson requires a training

Visual Basic	3	Jones	2	Williams	Technology is obsolete
MS SQL	2	Williams	1	–	–

A tabular way of information representation provides a quick overview on the knowledge distribution.

Retrospective meetings are often used in the agile methods in order to improve the process for the next iteration. It is a meeting held by a project team at the end of a project or process (in agile methods: iteration) to discuss what was successful about the project or time period covered by that retrospective, what could be improved, and how to incorporate the successes and improvements in future iterations or projects. Retrospective meetings assure that feedback from team members comes into development process, thus improving overall development quality. Such feedback may also be helpful during maintenance phase, where a team may learn on mistakes from the past. It might be a good strategy to document such decisions (e.g. in a WiKi) or some other knowledge database, so that these lessons-learned do not get lost.

Knowledge database may have several forms: *A personal knowledge database* contains persons and knowledge carried by them, the access to such database is usually open to everybody. However such database may become quickly out of date and itself produces too much maintenance overhead. *Wikis* may be used for information to be kept and exchanged, providing an easy and quick access to e.g. rules, experiences, decisions, etc. However Wikis may lack usability and scalability. They quickly become unmanageable.

Review is a process or meeting during which a software product is examined by project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval (IEEE 1028, Wikipedia). For more information c.f. Section 4.5.4.

Meetings / Groups can include so-called "team rounds", meeting dedicated to some special issue or topic and groups of interest. Meetings contribute to the distribution of the various knowledge types between team members and teams. It is a good way of keeping team members synchronised during some development and maintenance activities.

In-Team rotation is a practice, where different team members exchange their team roles or projects. Each team member will be familiar with the whole system (or sub system in case of larger systems) and, on demand, will be able to overtake tasks when other team members are unavailable or are overloaded at the moment. Though such "non specialists" are usually less efficient at implementing task beyond their profile, they still are able to execute these successfully during a reasonable time for task implementation.

Evaluation:

Development Phase:	Design / Implementation / Maintenance
Relevance, Automation:	Medium (Not automation specific, however, no domain limitation)
Relevance, Sustainability:	High (However, do not scale well over time)
Applicability:	Medium (These are simple practices, which are widely spread in the development community. However, their applicability might be restricted by legal issues or complexity and conditions of automation domain)

Tool:	–
Preventive / Reactive / Analytical:	Preventive / Reactive / Analytical
Formalization:	Informal
Perspective:	Not applicable
Abstraction level:	High
Benefit for sustainable systems:	Keep an overview of the knowledge state in organisation. Prevent unnecessary implementation and maintenance overhead caused by insufficiently informed team members. Eventually reduce dependency on "experts" in an knowledge area.

References:

- [Bommer C., Spindler M., and Barr V., Softwarewartung. dpunkt.verlag, 2008, Chapter 7] [BSB08] (book)
- You may also apply to section 3.3 "Historical Data, Data Mining" for information on data mining approaches (restoring information from various informational sources, e.g. code repositories and bug tracking systems).

4.5.2.2 Documentation Artefacts

Under documentation artefact we understand:

- Text documents (specifications, manuals, requirements specifications, etc.)
- Models (UML - Unified Modelling Language, DSM - Domain-specific modelling, etc.)

These artefacts are useful for performing activities during implementation and maintenance phases. Team members (specialists) may apply to them in order to obtain information about the system or system changes (design and/or implementation changes) required to perform their activities. If those artefacts are regularly updated, they help to reduce overhead connected to the system understanding. For long living systems in automation domain sufficient system documentation is highly relevant, as such systems are very complex and initial team members are usually unavailable for support.

Clearly, the documentation itself has to be regularly maintained, and this produces additional cost and overhead. It might even seem that these costs and overhead generally do not pay off. However, several studies provided as references in this subsection tend to prove positive result of investments into documentation during the system's maintenance phase [CRR09, DAB08, AHGT06, TH03, Try97]. Several selected statements from these surveys (for detailed information please apply to the surveys):

- "The results indicated that having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20 percent"
- "The subjects in the UML group had on average a practically and statistically significant 54% increase in the functional correctness of changes (p=0.03), and an insignificant 7% overall improvement in design quality (p=0.22)"
- "There are clear benefits to be derived from using UML e.g., traceability from functional requirements to code"
- "The subjects performed better with DSM (in investigated maintenance activities) for each dependent variable, although the subjects had extensive UML training but only brief experience with DSM."

More information can be found in the brief summary for each provided reference.

Evaluation:

Development Phase:	Design / Implementation / Maintenance
Relevance, Automation:	High (Although not automation specific, documentation help to deal with the domain complexity)
Relevance, Sustainability:	High (Documentation supports knowledge about the system over long period of time)
Applicability:	Medium (Created additional overhead, especially connected with maintaining documentation)
Tool:	–
Preventive / Reactive / Analytical:	Preventive
Formalization:	Formal / Informal

Perspective:	Positive
Abstraction level:	Not applicable
Benefit for sustainable systems:	Reduces overhead of maintenance activities, improves system understandability

References:

- [E. Tryggeseth. Report from an Experiment: Impact of Documentation on Maintenance. Empirical Software Eng., vol. 2, 201–207, 1997] A controlled experiment investigated how access to textual system documentation (the requirements specification, design document, test report, and user manual) helped when performing maintenance tasks. The results indicated that having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20 percent. The subjects who had the documentation available also showed better understanding and a more detailed solution to how the change can be incorporated as compared to those who had only the source code available. The results also suggested that there is an interaction between the maintainer's skill (as indicated by a pretest score) and the potential benefits of the system documentation: The most skilled maintainers benefited the most from the documentation. (Adapted from journal paper [DAB08]) [Try97] (journal paper)

- [W.J. Dzidek, E. Arisholm, and L.C. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. IEEE Transactions on Software Engineering, v 34 , Issue:3, 2008] Controlled experiment that investigates the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of a real, non-trivial system, using professional developers as subjects, working with a state-of-the-art UML tool during an extended period of time. The subjects in the control group had no UML documentation. In this experiment, the subjects in the UML group had on average a practically and statistically significant 54% increase in the functional correctness of changes (p=0.03), and an insignificant 7% overall improvement in design quality (p=0.22) - though a much larger improvement was observed on the first change task (56%) - at the expense of an insignificant 14% increase in development time caused by the overhead of updating the UML documentation (p=0.35). This experiment confirms that the presence of additional documentation in UML form gives the developers a better understanding of the system (via better correctness results). Authors state that they obtained similar results by using different measurements, both with trained students and professionals and systems of widely varying size, and that they are confident that UML will bring practically significant benefits under a large number of conditions.[DAB08] (journal paper)

- [B. Anda, K. Hansen, I. Gullesen, and H.K. Thorsen. Experiences from Using a UML-Based Development Method in a Large Organization. Empirical Software Eng., vol. 11, 555 – 581, 2006] Evaluation whether using UML is cost effective in a realistic context for a large project. The participants in the case study acknowledged that, despite some difficulties (e.g., the need for adequate training), there are clear benefits to be derived from using UML (e.g., traceability from functional requirements to code). All of the subjects in the UML group found the diagrams to be useful and traceability was enhanced. [AHGT06] (journal paper)

- [S. Tilley and S. Huang. A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding. Proc. ACM International Conference on Design of Communication 2003, ACM SIGDOC '03, pages 184–191, 2003] Qualitative efficiency of UML diagrams in aiding program understanding. Fifteen subjects whose UML expertise varied (six beginners, eight intermediate, and one expert) had to analyse a series of UML diagrams (with access to code) acknowledged that, UML's efficacy in supporting program understanding is limited by 1) unclear specifications of syntax and semantics in some of UML's more advanced features, 2) spatial layout problems

(e.g., large diagrams are not easy to read), and 3) insufficient support for representing the domain knowledge required in understanding a program. [TH03] (conference paper)

- [Lan Cao, B. Ramesh, and M. Rossi. Are Domain-Specific Models Easier to Maintain Than UML Models? Software, IEEE, Volume: 26 , Issue: 4, 19 – 21, 2009] Investigates if DSM (domain specific models) improve the maintenance performance of designers, compared to general-purpose modeling using UML. Authors investigated how each type of modeling language affects model comprehension, the correctness of changes, and the degree of changes made during a maintenance task. They also assessed the accuracy with which designers understand model syntax and model semantics. Based on results authors suggest that the subjects performed better with DSM for each dependent variable, although the subjects had extensive UML training but only brief experience with DSM. The DSM models' correctness score was about 20 percent higher than the UML models' score. The degree of changes in DSM was much smaller than in UML; UML diagrams required nearly twice the number of steps.
[CRR09] (journal paper)

- [Antony Tang, Jun Han, and Rajesh Vasa. Software Architecture Design Reasoning: A Case for Improved Methodology Support. Software, IEEE, 26, Issue: 2, 43 – 49, 2009] educe maintenance overhead caused by wrong design decisions. Software Architecture Design Reasoning proposes capturing and recording the reasoning behind software architecture design, which can encourage architects to more carefully consider design decisions and better support future maintenance. [THV09] (journal paper)

4.5.3 Consistency between artefacts

Artefacts present different views on a system, they support understandability of it and are used (either modified, e.g. code, or support modification, e.g. architectural models) during the maintenance phase. Preventing inconsistency between various system artefacts contributes to improving system understandability and preventing mistakes at system maintenance.

Artefact types:

- Documentation (design documents, requirement specifications, guides, manuals, reports, etc.)
- Models, polymeric views
- Source code
- Wiki
- FAQs

However, keeping artefacts consistent is a challenge. Inconsistent artefacts may lead to mistakes through false understanding of the system and implementation of bugs or wrong functionality. Inconsistent documentation, therefore, increases time and overhead required for evolving the system. Another aspect of inconsistency between artefacts is architecture violations. This means that the implementation of the system does not follow the initial architectural plan. In such cases violations either shall be prevented or architectural models shall be updated accordingly to the code.

In the following document, we provide a brief state-of-the-art overview for the following aspects:

- Tracing architecture violations - this is called compliance checking and can be found in the section
- Tracing documentation and code consistency

These approaches can be complemented by those presented in Section 3.4.3, e.g. to create initial versions or to contribute parts of the documentation.

4.5.3.1 Architecture Compliance Checking

Goal: Architecture compliance checking has the following goals:

- Identification of deviations between the intended architecture and the implemented architecture
- Identification of changes to the architectural design description during implementation and maintenance activities
- Detection of the structural violations

Architecture compliance checking contributes to the prevention of architecture decay (erosion) and drift between implementation and structural view. As system structure remains conforming to the describing artefacts, the overall understandability of the system remains on a more satisfactory level. This pays off during further maintenance activities, as it helps to reduce costs and overhead connected to them. A maintained system structure additionally contributes to the overall system quality and maybe even to the system performance. Beside that, the automation domain is controlled by strict regulations and standards. So, compliance checking might also contribute to not violating these regulations.

A Comparison of Static Architecture Compliance Checking Approaches (Knodel et al.)

Short Description: Guidance on when to use which static architecture compliance checking approach. Architecture compliance checking is an approach to detect architectural violations (i.e., deviations between the intended architecture and the implemented architecture). The paper compares three static architecture compliance checking approaches (reflection models, relation conformance rules, and component access rules) by assessing their applicability in 13 distinct dimensions. The main differences between analysed approaches concern the dimensions defect types, maintainability, transferability, multiple view support, and restructuring scenarios. Considering robustness of the architecture compliance checking approach with respect to code evolution, relation conformance rules seem to fit the best. Authors use TSAFE - a prototype of the Tactical Separation Assisted Flight Environment specified by NASA Ames Research Center - for their case study.

Evaluation:

Development Phase:	Implementation /Maintenance
Relevance, Automation:	Medium (Case study executed on a prototype from Aerospace domain)
Relevance, Sustainability:	Medium
Applicability:	Low (A study of approaches, more theoretical purpose)
Tool:	–
Preventive / Reactive / Analytical:	Preventive / Reactive
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Medium
Benefit for sustainable systems:	Results may help selecting a more suitable static approach for the architecture compliance checking. It may prevent architectural erosion and reduce maintenance overhead.

References:

- [Knodel J. and Popescu D. A Comparison of Static Architecture Compliance Checking Approaches. The Working IEEE/IFIP Conference on Software Architecture, 2007. WICSA '07. Page 12, 2007] [KP07] (conference paper)

Static Architecture Conformance Checking - An Illustrative Overview (Passos et al.)

Short Description: Compares and illustrates the use of three static architecture conformance techniques:dependency structure matrices (Lattix Inc's Dependency Manager), source code query languages (Semmle's .QL), and reflection models (Fraunhofer IESE's SAVE). To highlight the similarities and differences between the three techniques, it describes how some of their available supporting tools can be applied to specify and checks architectural constraints for a simple personal information management system. The authors provide following conclusions:

Criteria	DSM/LDM	SCQL/.QL	RM/SAVE
Expressiveness	Limited (only can-use, cannot-use constraints)	High (Datalog semantics)	Medium (regular expressions, but not subtypes)
Abstraction level	Medium (based on package hierarchy)	Low (based on code queries)	High (models provided by architects)
Ease of application	Medium (requires design rules for each constraint)	Medium (requires queries for each constraint)	Medium (requires mapping between models)
Architecture reasoning and discovery	High (DSM help to reveal architecture patterns)	Medium (warning, tables, graphs, charts, and tree maps)	Limited (only marks in the reflection model)

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (Not automation specific, no limitation for domain)
Relevance, Sustainability:	Medium (Detects violations, therefore supports sustainability. Manual actions required)
Applicability:	High (Techniques are applied in industry)
Tool:	Semmle's .QL: semmle.com, Lattix LDM: lattix.com, SAVE (link to Fraunhofer IESE): www.iese.fraunhofer.de
Preventive / Reactive / Analytical:	Reactive / Analytical
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Comparative evaluation of the architecture conformance techniques and tools. Such techniques may prevent architectural erosion and reduce maintenance overhead.

References:

- [Passos L., Terra R., Diniz R., Valente M. T., and Mendon N. Static Architecture Conformance Checking - An Illustrative Overview. Software, IEEE, PP , Issue: 99, 2009] [PTD+09] (journal paper)

Lightweight prevention of architectural erosion (O'Reilly et al.)

Short Description: The authors present a lightweight approach to the control of architectural erosion. It covers the representation of an architectural description and the management of alignment between description and implementation during system evolution. A prototype support tool, ArchAngel, is introduced. This tool is meant to:

- support the building and maintenance of simple architectural descriptions;
- support the linking of an architectural description to an implementation;
- be proactive in determining whether or not an evolving implementation conflicts with the defined dependencies;
- notify stakeholders (software engineers and architects) of inconsistencies that are detected (no assumptions have been made about how these stakeholders will use this information).

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	High (Concerns prevention of the top relevant problems for the automation domain)
Relevance, Sustainability:	High (Concerns prevention of the top relevant problems for the sustainability)
Applicability:	Low (Inapplicable, no link to prototype tool found, and unclear if the tool was further developed)
Tool:	–
Preventive / Reactive / Analytical:	Preventive / Reactive / Analytical
Formalization:	Formal
Perspective:	Positive / Negative
Abstraction level:	Medium / High
Benefit for sustainable systems:	Early recongnition and prevention of inconsistency and architectural erosion, thus, lower costs and maintenance overhead

References:

- [C. OReilly, P. Morrow, and D. Bustard. Lightweight prevention of architectural erosion. Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings, pages 59 – 64, 2003] [OMB03] (conference paper)

Constructive architecture compliance checking — an experiment on support by live feedback (Knodel et el.)

Short Description: Detection of the structural violations to prevent an architecture decay, prevention of the drift between the implementation and the structural view. It uses analytical reverse engineering technology (architecture compliance checking, automated as a variant of the SAVE - Software Architecture Visualization and Evaluation - tool) for monitoring the modifications made by developers. The new variant "SAVE LiFe" is integrated into the Eclipse development environment (IDE) and enables the live feedback for multiple developers. When a structural violation is detected, the particular developer receives feedback allowing prompt removal of the violations. These shall prevent architecture decay. A case study was performed with the help of 6 development teams. Three teams supported by the live compliance checking inserted about 60% less structural violations into the architecture than did the three other development teams.

Evaluation:

Development Phase:	Implementation / Maintenance
Relevance, Automation:	Medium (Not automation specific, however, no domain limitation)
Relevance, Sustainability:	High (Concerns architecture erosion problem, which is common for sustainable systems)
Applicability:	Low (Too few information about the developed tool is provided in the paper, tool itself was not found for download)
Tool:	– (Link to SAFE LiFe not available; link to Fraunhofer IESE, SAVE: www.iese.fraunhofer.de)
Preventive / Reactive / Analytical:	Preventive
Formalization:	Not applicable (Not enough information)
Perspective:	Positive / Negative
Abstraction level:	Low / Medium / High
Benefit for sustainable systems:	Prevent architecture erosion by notifying developers about architectural violations at real-time during their activities

References:

- [J. Knodel, D. Muthig, and D. Rost. Constructive architecture compliance checking – an experiment on support by live feedback. IEEE International Conference on Software Maintenance, 2008. ICSM 2008, pages 287 – 296, 2008] [KMR08] (conference paper)

4.5.3.2 Documentation and Code Consistency

Goal: Establish and maintain links between the source code and free (natural language) text documents. Support software evolution through representation of concerns in code.

Documentation and code consistency checking contributes to prevention of drift between artefacts and actual implementation of the system. If the actual implementation of the system remains conform to the describing artefacts, the overall understandability and traceability of the system improves. This pays off during further maintenance activities, as it helps to reduce costs and overhead connected to them (caused through overhead by understanding the system). Section 4.5.2.2 contains several studies on positive influence of documentation on the implementation phase.

Beside advantages during the maintenance phase, linking code and documentation (and their consistency) may contribute to requirement tracing, reuse of existing implementation parts and change impact analysis.

Recovering Code to Documentation Links in OO Systems (Antoniol et al.)

Short Description: An approach to establish and maintain traceability links between the source code and free text documents (documentation is expressed informally, stochastic language models are used for document's estimation). Supposes that programmers use meaningful names for program's items, such as functions, variables, types, classes, and methods and assumes that the application-domain knowledge that programmers process when writing the code is often captured by the mnemonics for identifiers; therefore, the analysis of these mnemonics can help to associate high level concepts with program concepts, and vice-versa. This approach is demonstrated on software written in C++ in order to trace classes to manual's sections. The process is automated (tool supported), it basically maps classes to the ordered list of manual sections.

Evaluation:

Development Phase:	Maintenance
Relevance, Automation:	Medium ((Not automation specific, however, no domain limitation)
Relevance, Sustainability:	Low (Long living systems tend to lack or to have outdated documentation, therefore there might be a few use for recovering links between code and documentation)
Applicability:	Low (Although, this approach is fully automated, link to the tool was not found)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Not applicable
Benefit for sustainable systems:	Support developers at exploring legacy systems

References:

- [G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering Code to Documentation Links in OO Systems. Proceedings. Sixth Working Conference on Reverse Engineering, pages 136 – 144, 1999] [ACLM99]

Recovering traceability links between a simple natural language sentence and source code using domain ontologies (Yoshikawa et al.)

Short Description: Establish and maintain links between the source code and natural language text documents. Proposes an ontology-based technique for recovering traceability links between a natural language sentence specifying features of a software product and the source code of the product. If a software product is insufficiently documented (or documentation is out of date), this approach may be used to automatically detect code fragments associated with the functional descriptions written in the form of simple sentences. The relationships between source code structures and problem domains are important. The knowledge of the problem domains is modelled as domain ontologies. A provided case study demonstrates better results than without ontologies.

Evaluation:

Development Phase:	Maintenance
Relevance, Automation:	Medium (Not automation specific, however, no domain limitation)
Relevance, Sustainability:	Low (Long living systems tend to lack or to have outdated documentation, therefore there might be a few use for recovering links between code and documentation)
Applicability:	Low (The approach is still at the beginning of development)
Tool:	–
Preventive / Reactive / Analytical:	Analytical
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Medium
Benefit for sustainable systems:	Support developers at exploring legacy systems

References:

- [T. Yoshikawa, S. Hayashi, and M. Saeki. Recovering traceability links between a simple natural language sentence and source code using domain ontologies. IEEE International Conference on Software Maintenance, 2009. ICSM 2009. Pages 551 – 554, 2009] [YHS09]

Representing Concerns in Source Code (Robillard)

Short Description: Support software evolution through representation of concerns in code. Description of concerns, representing program structures and linked to source code that can be produced cost-effectively during program investigation activities. This can help developers perform software evolution tasks more systematically, and on different versions of a system. It makes possible to precisely define the notion of inconsistency between a concern graph and the corresponding source code and automatically detect and repair inconsistencies between a description of source code and an actual code base. The approach is tool-supported: FEAT, an Eclipse plug-in for locating, describing, and analysing concerns in source code.

Evaluation:

Development Phase:	Implementation
Relevance, Automation:	Medium (Not automation specific, however, no domain limitation)
Relevance, Sustainability:	Medium (Long living systems tend to lack or to have outdated documentation)
Applicability:	Medium (Has a tool, however, it's usability was not analysed)
Tool:	FEAT Tool: http://www.cs.mcgill.ca/~swevo/feat/
Preventive / Reactive / Analytical:	(Preventive / Reactive / Analytical)
Formalization:	Formal
Perspective:	Negative
Abstraction level:	Low / Medium
Benefit for sustainable systems:	Support maintenance activities at insufficiently documented systems, thus, reducing overhead and costs

References:

- [Robillard. Representing Concerns in Source Code. PhD thesis, 2003] [Rob03] (PhD Thesis)

4.5.4 Quality Assurance Strategies

General quality assurance strategies contribute to higher system quality and, therefore, potentially lower maintenance and system evolution costs and overhead. Examples of the quality assurance strategies are:

- Testing (Unit testing, integration testing, etc.)
- Reviews (Code reviews, inspections, pair programming, etc.)

There is a tool supportt for the quality assurance, i.e. SISSy, ISIS, Checkstyle, Findbugs, etc.

Testing: detailed overview of the testing techniques is out of scope of this document. However has a great impact on evolvability of (long living) systems. Systems having a high test coverage usually have less bugs that are found during maintenance phase. This leads to reduced (fixing) changes to the system during maintenance and, thus, fewer possibilities that lead to erosion of software.

Reviews: Reviews are parts of quality assurance strategies and have the same benefits as testing. There are several review types, for example the following:

- *Code review* is systematic examination (often as peer review) of computer source code.
- *Pair programming* is a type of code review where two persons develop code together at the same workstation.
- *Inspection* is a formal type of peer review where the reviewers are following a well-defined process to find defects.
- *Walkthrough* is a form of peer review where the author leads members of the development team and other interested parties through a software product and the participants ask questions and make comments about defects.
- *Technical review* is a form of peer review in which a team of qualified personnel examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.
- *Fagan review* is a structured process of trying to find defects in development documents such as programming code, specifications, designs and others during various phases of the software development process.
- *Semantic analysis* is tracking variable/function/type declarations and type checking.
- *Control-flow analysis* is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

SISSy / ISIS: are examples of tools that can be used as a part of quality assurance strategies. For more information on SISSy and ISIS apply to: SISSy Section 3.4.3.2 and ISIS Section 3.4.3.3.

References:

- [Balzert H. Lehrbuch der Software-Technik - Software-Entwicklung. Spektrum-Akademischer Vlg, 2000 (in German)] [Bal00] (book)
- [Ludewig J. and Lichter H. Software Engineering. Dpunkt Verlag, 2006] [LL06] (book)
- Projects (tools):

  SISSY `http://sissy.fzi.de`,

  ISIS `http://www.andrena.de/node/160/`.

4.5.5 Team Organization Strategies

One of the goals of optimizing team organization (beside reducing the costs) is to increase quality of the developed system that has an impact on further evolution and migration complexity. A detailed overview of the team organization strategies do not contribute directly to the goals of this project. However, it is important to realize that a wrong organization of a development team may produce a negative impact on quality and evolution phase of the system life cycle.

4.5.6 Development Environment Strategies, Virtualisation

The development environment and its evolution exhibit an impact on the later maintenance of long living systems. Therefore, the goal would be to explicitly consider reducing dependency to the evolution of development environment. For example ABB uses Microsoft Technologies for system development, which leads to the need of co-evolution with them.

This problem may be solved partially through the virtualisation of the development environment. This helps to retain the development environment as used to on top of evolving hardware technology.

Virtualization is a broad term which encompasses a number of different technologies. Under virtualization we understand the "separation of a resource or request for a service from the underlying physical delivery of that service".

Virtualization may not yet solve the specific hardware problems, however it can be successfully applied for resolving of the following issues at ABB:

- enabling application migration
- reducing footprint
- maintaining development environment
- reducing installation effort

Application of virtualization solutions at ABB is only possible under consideration of the standards, mentioned in the Section 2.5 "Relevant standards".

Detailed overview of the development environment strategies is out of scope of this document.

References:

- [Barnett R. J., Irwin B. V.. Virtualized Systems and their Performance: A Literature Review. 2007] [BI07] (conference paper)
- [Menascé D. A.. Virtualization: Concepts, Applications, and Performance Modeling. White Paper, George Mason University, 2005] [Men05] (white paper)
- [Wolf C. and Halter E. M.. Virtualization: from the desktop to the enterprise. Apress; 1 edition, 2005] [WH05] (book)

4.6 Management Strategies: Make or Buy Decision Support

Certain decisions regarding infrastructure technology are met at the management level. These are decisions concerned with development policy, such as (Section 4.6):

1. Buying third party solutions (COTS, Commercial-Off-The-Shelf)
2. Using Open-source Software Solutions (adaptation)
3. Implementing own solutions

These management decisions influence the evolution phase of software, as they have impact on the system's maintenance. So for example buying third party solutions (or using open source solutions) may reduce development costs, however may also create dependencies to the evolution of external software. Virtualization may help keeping outdated technology (required for system maintenance) available or in some cases reduce dependency to the underlying software. However, it creates additional costs, requires special knowledge and maintenance itself.

The goal is to reduce dependency to the evolution of external software, and, from another side, reduce overhead of the development of own software.

Term "Off-the-shelf" (OTS) is used to summarise external development options, such as Commercials-Off-The-Shelf (COTS) or open source software (OSS). Decisions either to implement or to buy (adapt) software have advantages and disadvantages. So buying third party solutions (or using open source solutions) may reduce development costs, however may also create dependencies to the evolution of external software. Implementation of own software might be more expensive, however the evolution of such software is depending on organisation itself.

This provides a brief overview for the following aspects, concerning make or bye decisions:

- Risks connected with the use of COTS
- Selection of the COTS, integration of the COTS usage into the process
- Trade-off between Make or Buy decision, advantages and disadvantages, COTS and Open Source
- Maintenance of COTS systems

This section has an introductory purpose, the majority of references presents a discussion of the topic.

4.6.1 Risks, Selection and Integration of the COTS into the process

The process of selection of OTS (COTS or OSS) may be either formal or informal:

- Informal ways include using in-house expertise and Web-based search engines. Such ways may be more suitable in case of the limited number of OTS candidates available, or the company's pre-existing, long-term partnership with a specific provider.
- Formal ways are for example direct assessment, compatible COTS component selection (CCCS) or domain-based COTS product selection method. However, there is few empirical evaluation of formal processes for selecting OTS components, the cost benefits and preconditions of using a formal process are often unclear.

OTS may be selected during design, implementation and maintenance phases. The study [LCB+09] (journal paper) points that OTS components are mainly selected based on architecture compliance instead of function completeness and used without modifications.

Integration of OTS into the system is connected to certain effort. This effort may be estimated by using personal experience of persons responsible for integration (however, they are usually inaccurate) or using a formal effort estimation approach (e.g. tool COCOT, http://csse.usc.edu/csse/research/COCOTS/index.html or EPIC process, http://www.sei.cmu.edu/).

Usage of OTS is connected to certain risks, these risks concern OTS selection and integration phases, as well as system maintenance phase. For example, during OTS selection and integration there may be the following risks:

- Wrong OTS was selected (at the end, it may become more expensive then self-implementation)
- Underestimation of integration effort
- High learning curve required to adapt an OTS
- Lack of evaluated methods for OTS selection and integration
- OTS are not sufficiently compatible with the deployment environment of the system
- Different quality practices of a vendor and purchasing organisation

Examples of risks during the maintenance phase:

- Asynchronous update cycles, rapid and asynchronous changes
- The new OTS version is incompatible with the system (incl. its environment or legal regulations)
- The OTS provider stops its support
- Licensing, legal issues and new standards
- Hard to estimate if a defect is inside or outside the OTS borders

References:

- [Elisabeth Hansson and Goran V. Grahn. One Global COTS-Based Systemto Replace 20+ Local Legacy Systems. Lecture Notes in Computer Science,COTS-Based Software Systems, 3412/2005, 144 – 145, 2005] Presentation of the technical challenges and lessons learned within the project, replacing 20+ different IT systems, both in-house developed and bought packages, with one COTS-based IT system. Because of long living property of the new system, the COTS based solution had to be open, flexible and scalable over time. Another key part of the architecture required in their solution was integration to existing systems. [HG05] (outline)
- [Francis L. Lifetime procurement - Look deep for dependability. Electronics Systems and Software, 4, Issue:6, 22 – 25, 2006] Article on the obsolescence planning in the military area, concerns hardware components. Points to problems also relevant in the automation domain (e.g. different life cycles of systems and COTS). [Fra06] (journal paper)
- [Jingyue Li, Conradi R., Odd P., Slyngstad O., Torchiano M., Morisio M. and Bunse C. Validation of New Theses on Off-the-Shelf Component Based Development. Software Metrics, 2005. 11th IEEE International Symposium, pages 26 – 26, 2005] Survey on the OTs usage in the praxis. The supported theses are: 1) OSS components were mainly used without modification in practice, custom code mainly provided additional functionality; 2) formal OTS selection processes were seldom used. The unsupported theses are: 1) standard mismatches were more frequent than architecture mismatches; 2) OTS components were mainly selected based on architecture compliance instead of function completeness [LCS+05] (conference paper)

- [Jingyue Li, Conradi R., Bunse C., Torchiano M., Slyngstad O., and Morisio M.. Development with Off-the-Shelf Components: 10 Facts. Software, IEEE, 26 Issue:2:, 80 – 87, 2009] Article on the survey's results from the year 2008 by the same authors (see [LCS+05] listed in this section) [LCB+09] (journal paper)

- [Land R., Blankers L., Chaudron M., and Crnkovic I. COTS Selection Best Practices in Literature and in Industry. Lecture Notes in Computer Science, High Condence Software Reuse in Large Systems, pages 100 – 111, 2008.] Literature survey of the software COTS component selection methods, followed by a metamodel consolidating the activities and practices of these methods. Provides recommendations which will enable organizations to identify suitable practices when designing a customized selection processes. [LBCC08] (journal paper)

  [Jingyue Li,Bunse C., Torchiano M., Slyngstad O., and Morisio M.. A State-of-the-Practice Survey of Risk Management in Development with Off- the-Shelf Software Components. IEEE Transactions on Software Engineering,34 , Issue:2, 271 – 286, 2008] An international survey on risk management in software development with off-the-shelf (OTS) components. Analysed data from 133 software projects in Norway, Italy, and Germany. [LST+08] (journal paper)

- [Becker S. Cost Model, Decision Support and Selection Process for COTS. Dimploma thesis, Carl von Ossietzky Universitaet Oldenburg, 2003] (in german) [Bec03] (dimploma thesis)

- [Kotonya G., Hutchinson J.. Analysing the Impact of Change in COTS-Based Systems. Lecture Notes in Computer Science, COTS-Based Software Systems, 3412/2005, 212 – 222, 2005] An approach to to impact analysis in COTS-based system. It is based on the use of a COTS component oriented development process and an architecture description language (ADL) for documenting component system architectures. [KH05] (journal paper)

- [Lewis P. et al.. Lessons Learned in Developing Commercial Off-The-Shelf (COTS) Intensive Software Systems. Technical report, The Federal Aviation Administration (FAA), 2000] Tech report on topics: problems with vendors, leverage in gaining vendor cooperation, need for flexibility in defining requirements, importance of operational demonstrations, assessment of specific attributes, life-cycle issues, COTS integrator experience, need for technology watch to keep up with vendors, interface to legacy systems, impacts of volatility during development, vendor management. [Lew00] (Tech report)

- [Lipson H. F., Mead N. R., and Moore A. P.. Can We Ever Build Survivable Systems from COTS Components? Lecture Notes in Computer Science, Advanced Information Systems Engineering, 2348/2006, 216 – 229, 2001] Describes a risk-mitigation framework for deciding when and how COTS components can be used to build survivable systems. [LMM01] (journal paper)

- [Meine van der Meulen, Riddle S., Strigini L., and Jefferson N.. Protective Wrapping of Off-the-Shelf Components. Lecture Notes in Computer Science, COTS-Based Software Systems, 3412/2005, 168 – 177, 2005] Discusses the use of wrappers to improve the dependability – i.e., "non-functional" properties like availability, reliability, security, and/or safety – of a component and thus of a system. Wrappers can improve dependability by adding fault tolerance, e.g. graceful degradation, or error recovery mechanisms. [vdMRSJ05] (journal paper)

- [Leung K., Leung H. On the eficiency of domain-based COTS product selection method. Information and Software Technology, Volume 44, Issue 12, 703–715, 2002] Discussion of the efficiency of domain-based COTS product selection method [LL02] (journal paper)

4.6.2 Maintenance of COTS

Maintenance of the COTS systems is a complicated topic. The cost to maintain such systems often is equal or even exceeds that of developing custom software. The maintenance activities include:

- Bug-fixing
- Licensing, new standards
- Version and technology upgrades
- Support of obsolete technologies

One of the most common troubles is asynchronous update cycles of COTS and long-living systems in automation domain. Such systems usually have long update cycles equal to several years (coupled to maintenance of underlying hardware), whether COTS might have an update cycle of several weeks.

Examples of risks during the maintenance phase (duplicate from the previous section):

- Asynchronous update cycles, rapid and asynchronous changes
- The new OTS version is incompatible with the system (incl. its environment or legal regulations)
- The OTS provider stops its support
- Licensing, legal issues and new standards
- Hard to estimate if a defect is inside or outside the OTS borders

References:

- [Reifer D., Boehm B., Basili V. and Clark B.. Eight Lessons Learned during COTS-Based Systems Maintenance. IEEE Software, 20, 94 – 96, 2003] Results of the empirical study [RBB+03] (journal paper)
- [Vigder M. R. and Dean J.. Maintaining COTS-Based Systems. Fifth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'06), pages 11 – 18, 2000] Maintenance of COTS-Based systems and its difference from custom-built systems [VD00] (conference paper)
- [Clapp J. A. and Taub A.E.. A Management Guide to Software Maintenance in COTS-Based Systems. MITRE, Center for Air Force C2 Systems, Bedford, Massachusetts, 1998] Planning of maintaining COTS software products in COTS-based systems. Discusses issues and risks in using COTS over the life cycle and how to control them [CT98] (tech report)
- [Reifer D., Basili V. , Boehm V., and Clark B. COTS-Based Systems: Twelve Lessons Learned about Maintenance. Lecture Notes in Computer Science, COTS-Based Software Systems, 2959/2004, 137 – 145, 2004] Lists activities specially connected to the use of COTs during the maintenance process. It describes learned lessons about the maintenance of systems containing COTs and risks connected to it [RBBC04] (journal paper)

4.6.3 Trade-off between Make or Buy decision, COTS and Open Source

There is a trade-off between these options, so called "Make or Buy Decision". In section 4.5 of [RH09] (book) authors list several advantages and disadvantages of such make or buy decisions. Decisions between make or buy affect the reusability of own or external development results.

Implementation of the own software ("Make") has the following properties:

- (positive) It leads to own know-how and ownership of solutions
- (negative)It requires resources like time, money and qualifications/skills/know-how

Usage of the third party solution ("Buy") has the following properties:

- (positive) Less costs when using external
- (negative) Project management risks: Unknown effort for adaptation and integration, unknown fulfilment of performance and reliability requirements, unknown support of manufacturer with respect to integration or maintenance.

References:

- [Reussner R. and Hasselbring W.. Handbuch der Software-Architektur. dpunkt.verlag, 2. edition, 2009] [RH09] (book)
- [Jingyue Li, Bjornson F. O., Conradi R., and Kampenes V. B.. An empirical study of variations in COTS-based software development processes in the Norwegian IT industry. Empirical Software Engineering, Volume 11, Number 3, 433 – 461, 2006] Studies how to use and customize COTS-based development processes for different project contexts. Describes an exploratory study of state-of-the-practice of COTS-based development processes, briefly discusses: COTS-specific activities, risks, selection [LBCK06] (journal paper)
- [Giacomo P. D.. COTS and Open Source Software Components: Are They Really Different on the Battlefield? Lecture Notes in Computer Science, COTS-Based Software Systems, 3412/2005, 301 – 310, 2005] Discussion of the OSS and COTS, a guideline for the correct use of OSS within component-based systems [Gia05] (journal paper)
- [Mohagheghi P. and Conradi R.. Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering, Volume 12, Number 5, 471 – 516, 2007] Reviews systematic software reuse topic, assess the effects of software reuse in industrial contexts in the period between 1994 and 2005. Makes conclusions about benefits of reuse like: problem density (defect, fault or error), productivity, etc [MC07] (journal paper)
- [Holck J., Larsen M. H., and Pedersen M. K.. Managerial and Technical Barriers to the Adoption of Open Source Software. Lecture Notes in Computer Science, COTS-Based Software Systems, 3412/2005, 289 – 300, 2005] Discusses managerial and technical decisions for acquisition of OSS, compares COTS and OSS [HLP05] (journal paper)

Chapter 5

Related Surveys

In this chapter we summarise several publications that survey research and state of the art in software evolution and its related topics, e.g., software maintenance, reverse engineering, software refactoring, model-driven development.

[CI90] (journal paper) is about a taxonomy in the area of reverse engineering and design recovery. The paper addresses the problem of confusion about terminology in context of reverse engineering. The authors define and relate six terms: forward engineering, reverse engineering, redocumentation, design recovery, restructuring, and reengineering.

[Ben96] (journal paper) is about the past, present, and future of software evolution. In this paper the authors review progress in software evolution. They present a three-level approach to considering software evolution, in terms of the impact 1) on the organization, 2) on the process, 3) and on technology supporting that process. Progress is reported in all three levels. Moreover a proposal for an IEEE standard for maintenance processes is described. They encourage to think in terms of solutions instead of problems.

[BR00] (conference paper) is about a roadmap for software maintenance and evolution. The paper aims at describing a landscape for research in software maintenance, identifies key problems, solution strategies, and topics of importance. Trends and practices are projected forward by using a staged model of software evolution.

[CHK+01] (journal paper) is about types of software evolution and software maintenance. The paper proposes a (clarifying) redefinition of the types of software evolution and software maintenance. A classification of software evolution types is presented by distinguishing changes of (1) the software, (2) the documentation, (3) the properties of the software, and (4) the customer-experienced functionality. The paper provides a classified list of maintenance activities and a condensed decision tree as a summary guide to use the presented classification.

[MD01] (conference paper) is about future trends in software evolution metrics. The paper provides a classification of the various approaches that use metrics to analyse, understand, control and improve the software evolution process, and identifies topics that require further research.

[Men02] (journal paper) is about the state of the art (survey) in software merging. The paper provides a comprehensive survey and analysis of available merge approaches. The paper covers initial techniques based on textual merging, but also techniques that take syntax and semantics into account. Moreover operation-based merging is presented. After comparing the possible merge techniques, a number of important open problems and future research directions is discussed.

[MT04] (journal paper) provides a survey of software refactoring. The paper provides an extensive overview of existing research in the field of software refactoring. This research is compared and discussed based on a number of different criteria: the refactoring activities that are supported, the specific techniques and formalisms that are used for supporting these activities, the types of software artifacts that are being refactored, the important issues that need to be taken into account when building refactoring tool support, and the effect of refactoring on the software

process.

[BMZ+05] (journal paper) gives a taxonomy of software change that is based on characterizing the mechanisms of change and the factors that influence these mechanisms. The ultimate goal of this taxonomy is to provide a framework that positions concrete tools, formalisms and methods within the domain of software evolution. Such a framework would considerably ease comparison between the various mechanisms of change. It would also allow practitioners to identify and evaluate the relevant tools, methods and formalisms for a particular change scenario. As an initial step towards this taxonomy, the paper presents a framework that can be used to characterize software change support tools and to identify the factors that impact on the use of these tools. The framework is evaluated by applying it to three different change support tools and by comparing these tools based on this analysis.

[MWD+05] (conference paper) is about challenges in software evolution, that are considered important by the authors aiming to provide novel research directions in the software evolution domain.

[CP07] (conference paper) is about new frontiers of reverse engineering that presents an overview of the field of reverse engineering, reviews main achievements and areas of application, and highlights key open research issues for the future.

[FR07] (conference paper) is about a research roadmap for model-driven engineering (MDE) of complex software. The paper gives an overview of current research in MDE and discusses some of the major challenges that must be tackled in order to realize the MDE vision of software development. The authors argue that full realizations of the MDE vision may not be possible in the near to medium-term primarily because of some difficult problems, but attempting to realize the vision will provide valuable insights.

[KCM07] (journal paper) gives a survey and taxonomy of approaches for mining software repositories in the context of software evolution. The survey deals with those investigations that examine multiple versions of software artifacts or other temporal information. It presents work via four dimensions: the type of software repositories mined (what), the purpose (why), the adopted/invented methodology used (how), and the evaluation method (quality).

[TTBS07] (journal paper) is about the state of the art and future trends of empirical studies in reverse engineering. The authors' position is that the next stage of development for this discipline will necessarily be based on empirical evaluation of methods. The paper contributes a roadmap for the future research in the field with the goal to clarify the scope of investigation, to define a reference taxonomy, and to adopt a common framework for the execution of the experiments.

[Van07] (workshop paper) is about a research agenda for model-driven software evolution. The paper addresses the evolution of applications built by using model-driven development approaches and points out the need for consideration of multiple dimensions of evolution: regular evolution, meta-model evolution, platform evolution, and abstraction evolution. The authors identify problems and challenges for research in four research themes: model developments environments, "from model to code", "from code to models", and evaluation.

[GG08] (conference paper) is about the past, present, and future of software evolution. The paper discusses the definitions of evolution and maintenance terminology and compares Lehman's Laws with Staged Model (of maintenance and evolution) of Bennett and Rajlich. Moreover software evolution is compared with biological evolution. The authors present a roadmap with challenges and opportunities addressing six areas: 1) model building and empirical studies, 2) open source development, 3) evolutionary pressure and emergent design, 4) improving the collective memory of software developers, 5) the emergence of software "ecospheres", and 6) improved understanding of economic tradeoffs and risks.

Chapter 6

Management Summary

6.1 Categorisation of Approaches

This document presents a survey of analytical and proactive approaches for dealing with evolution of sustainable software systems. It aims at structuring the field of sustainable software engineering approaches.

In this survey document, we divided the presented approaches for identification and analysis of evolution problems (see Section 3) into three categories:

- Architecture analysis
- Software Comprehension using Historical Data
- Quality Indicators

Each category is briefly summarised in the following.

Architecture Analysis provides means for the development of understanding of the current state and investigation of the quality potential of software systems. It allows to derive strategic development perspectives for software systems. Approaches of this category are predominantly informal and manual. The most critical point of analytical approaches is that they are usually heavy-weight approaches since they require a large amount of participants to be involved in their application which typically represent different stakeholders of a software system. The challenge and change of architecture analysis lie in the creation of a common shared understanding of a software systems among all participants.

One of the most fitting approaches with respect to sustainability is ALMA (see Section 3.2.3). However the applicability of ALMA has limitations, e.g., there is no tool support available since the process is mostly informally described.

Software Comprehension using Historical Data has minor importance from the project perspective, since there are only few stringent implications from historical data on the current situation of a software system – especially on the architecture level. Approaches of this category are appropriate for problems with low granularity, e.g., probability of coding errors or the identifying of the right specialist for maintenance tasks and predicting which of a system's parts are most likely to change together. However, predictions based on historical data are less reliable, require special tools, large and well-maintained databases of historical data, and technical training, especially for result interpretation.

The absence of applicable solutions for the architecture-relevant analysis of historical data might be predominantly caused by lack of data to be analysed. Application of solutions for the

architecture-relevant analysis of historical data is only possible if historical data can be provided. In order to achieve relevant data a systematic collection and aggregation of data is necessary. Artefacts containing explicit historical information on the software architecture, if initially documented at all, are usually badly maintained and are mostly outdated. Architects should take care of this. Therefore, architecture artefacts have to be first recovered from system code stored in the version systems by using reverse engineering approaches.

Quality Indicators are mainly related to metrics and bad smells for detecting evolution problems (refer to Section 3.4 for more details about metrics and tools.). The overall question for approaches of this category is, whether evolution problems can be effectively detected by using metrics and bad smells which are typically based on the code representation of a software system. Literature proposes a huge amount of metrics, but their relationship to evolution and sustainability is difficult to prove or validate.

When establishing metrics and bad smell detections as a part of a software lifecycle, the awareness for low-granular and medium-granular problems in software systems can be significantly increased. The rationale behind quality indicators metrics and the awareness for bad smells should be an essential part of workforce training and basic knowledge for software developers to assist them in developing sustainable software systems.

There is substantial tool support for the calculation of software metric and bad smell detection available. The support by tools for identifying problems to sustainability can be considered as powerful. Furthermore, a high degree of automation can contribute in establishing quality assurance mechanisms for sustainable software. Implications of metrics and bad smell results have nevertheless to be derived manually since no direct implications for a software system exist (e.g. intended bad smell due to the application of a software pattern). As promising tools for calculation and monitoring of software metrics we recommend SISSy and Sotograph, see Section 3.4.1, for detection of bad smells we recommend SISSy and FindBugs, see Section 3.4.2, and for dependency analysis we recommend Lattix LDM, see Section 3.4.3.1.

A support for an automatic translation from detected problem patterns or bad smells into code improvements is not possible.

6.2 Decision Levels

The approaches for preventing and solving evolution problems in Section 4 address several decision levels. These levels are briefly sketched in the following. Each level should be reflected when aiming at sustainable software development an sustainable software systems.

Static Architecture and Software System Structure. The Static Architecture and Software System structure is crucial for achieving sustainable systems. Due to little automation and the sheer size of the design space of typical software systems, the appropriate usage of structural approaches is relying on human knowledge. Section 4.1 presents a selection of approaches that are highly relevant with respect to sustainability from the perspective of the static architecture and the system structure. These approaches represent basic knowledge that should be an essential part of workforce training. In the area of best practices we recommend the study of Modifiability Tactics and other common known design principle. Design Patterns should be common knowledge, especially patterns with influence on modifiability (e.g., Façade, Layers, Model-View-Controller, etc.). These approaches are good candidates for empirical validation using industrial case studies.

Reactive elimination of evolution problems deals with systematic removing of identified evolution problems, like bad smells, by performing small structured evolution steps. Refactoring approaches are necessary to keep a software system modifiable, to reduce complexity, and to

remove unnecessary dependencies. These approaches should be essential part of the development of sustainable software systems.

Variability Strategies surveys approaches that help to investigate the requirements of variability in software systems. Providing the appropriate degree of variability is important for sustainable software systems, since software systems should provide variability for parts that have to change during evolution. The kinds of sustainable systems, that we are addressing here, should in particular provide variability regarding the exchange and evolution of (underlying) technologies and therefore aim at separating business logic from technical layers. Project managers and software architects should use variability approaches to identify the technological dependencies and determine the demand for variation points. Since implementing variation points is always connected with additional costs this kind of analysis is necessary to keep the cost-benefit balance.

Automation of Software Development aims at increasing the consistency between artefacts (models, code, and documentation), increasing the productivity of software development, and reducing the time-to-market. This is achieved by rising the abstraction level of the code closer to the domain and by a better separation of domain knowledge and technology. Especially techniques like domain specific languages decouple domain knowledge from technical details and imply the hope to have persistent domain knowledge separated from more short-term technical knowledge. Studies which prove the sustainability of domain model are still lacking.

Architecture-Centric Model-Driven Software Development (AC-MDSD, see Section 4.4.3) is among the most promising approaches with respect to sustainable systems due to the explicit consideration of architectures and the explicit separation of platform and application aspects.

Development Process is another layer of preventing evolution problems. If a development process explicitly accounts for maintenance activities and quality assurance, establishing a sustainable software system is more likely. Hence, the choice of the right development process is suitable to reduce future evolution problems. There is no universal process solution supporting all types of the projects. Therefore, the development process should be selected depending on a project scope and environment conditions.

For example, lightweight processes, like agile development, are more suitable for smaller, rapidly changing projects or execution of short-term maintenance activities. However, agile methods in their pure form have rather negative impact on the later system maintenance, [KMSN+06, HYCM09] (conference papers), so such methods should be carefully used for the initial development of a software system. Currently available agile methods that support architectural modelling and consider further maintenance and evolution phases of a software system are not yet sufficiently developed and evaluated to an extend to be immediately applicable in the industries which rely on sustainability.

One of the interesting approaches for a detailed case study on influence of explicit architecture modelling on maintainability in agile methods could be an approach presented in [Mad10] (journal paper). It has a strong practice orientation and, according to the author, was aprobed in multiply projects in industry context. Another case study could be dedicated to the application of DCI architectures (not reviewed in this document because of insufficient data on its application) [CB10] (book).

Knowledge Management and Documentation. Sustainable software development subsumes knowledge management and transfer among generation of developers. Furthermore, the consistence of artefacts helps keeping knowledge. Empirical studies prove that sufficient documentation (also of the system architecture) pays off well during evolution and maintenance phases and improve the overall system quality. Clearly, any kind of documentation has to be maintained, otherwise the positive effect on reducing evolution problems will be lost.

Team Support. A development process should be complemented by corresponding team organisation strategies which reduce dependencies among subsystems and account for the structure of existing software systems. To support the maintenance tasks of software developers, appropriate environments can help which for example encapsulate outdated or legacy development environments and infrastructures in virtual machines. A detailed overview of these topics is out of scope of this document.

Software Infrastructure deals with decisions regarding the usage of third-party software. The decision on the usage of third-party software comprises the selection of criteria reflecting risk evaluation, maintainability properties, and an estimation on the ease of integrations with existing systems. Generally, there are two dimensions for the selection of third-party software. In the first dimension, one has to decide whether to i) buy readily available software or components from third party vendors or ii) let the software implement by third parties. The second dimension is to i) employ close source and black box software or to ii) use open source software.

6.3 Overall

This document identifies relevant approaches and techniques for sustainable software development. Aiming at sustainable software systems implies sustainable software development. The surveyed literature suggest a mixture of approaches for different stages of the software development process which are tailored on a per-project and on a per-domain base. Addressing sustainable software systems is a holistic task that cannot be mastered by single isolated means. Due to a strongly varying degree of tool support and maturity of proposed approaches in literature, automated approaches are suggested to be combined with trainings on partially automated and fully manual approaches to cover the spectrum of challenges for sustainable software systems. In order to provide operational decision support for software architects and project managers a guidelines document based on this document will be set up. It will serve as a kind of a cookbook for proactive evolution handling in sustainable software development.

Chapter 7

Information Sources

This chapter summarises the information source which were reviewed for contribution to the creation of this document.

7.1 Books

- Christopher Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, ISBN 0195019199, 1977
- Scott Ambler. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. Wiley, 2002
- Helmut Balzert. Lehrbuch der Software-Technik - Software-Entwicklung. Spektrum-Akademischer Vlg, 2000
- K. Beck and C. Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004
- Bommer C., Spindler, M., Barr, V., Softwarewartung, dpunkt.verlag, 2008
- Jan Bosch. Design and Use of Software Architectures Adopting and evolving a product-line approach. Addison-Wesley, 2000
- Frank Buschmann. A system of patterns. Wiley, repr. edition, 2001
- William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley and Sons, Inc., New York, NY, USA, 1998
- Paul Clements, Felix Bachmann, Len Bass, and David Garlan. Documenting software architectures: Views and beyond (sei series in software engineering). 2002.
- Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming. Addison-Wesley, 2000
- Paul Clements, Rick Kazman, and Mark Klein. Evaluating software architectures. Addison-Wesley, 4. print. edition, 2005
- Paul Clements and Linda Northrop. Software Product Lines Practices and Patterns. Addison Wesley, 2002
- A. Cockburn. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Longman, Amsterdam, 2004

- James Coplien and Gertrud Bjornvig. Lean Architecture: for Agile Software Development. John Wiley and Sons; Aufage: 1, 2010
- R. Dumke and F. Lehner. Software-Metriken Entwicklungen, Werkzeuge und Anwendungsverfahren. Deutscher Unversitaets-Verlag, 2000
- Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999
- Norman E. Fenton and Shari Lawrence Peeger. Software Metrics A Rigorous and Practical Approach. PWS Publishing Company, 1997
- Gamma E., Nackmann, L., Wiegand, J., Eclipse Modeling Project A Domain-Specific Language (DSL) Toolkit, Addison-Wesley, 2009
- Erich Gamma, Lee Nackmann, and John Wiegand. eclipse Modeling Project A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, 2009
- Jack Greenfield and Keith Short. Software Factories Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004
- Brian Henderson-Sellers. Object-oriented Metrics Measures of Complexity. Prentice Hall, 1996
- Ralph Johnson, John Vlissides Erich Gamma, Richard Helm. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995
- Stephen H. Kan. Metrics and Models in Software Quality Engineering (2nd ed.). Addison-Wesley Longman, 2002
- Jochen Ludewig and Horst Lichter. Software Engineering. Dpunkt Verlag, 2006
- Robert C. Martin. Clean Code A Handbook of Agile Software Craftmanship. Prentice Hall, 2009
- Stephen J. Mellor and Marc Balcer. Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- Mens T., Software Evolution, Springer Berlin Heidelberg, 2008
- S. R. Palmer, M. Felsing, and S. Palmer. A Practical Guide to Feature-Driven Development. Prentice Hall International, 2002
- Reussner R., Hasselbring, W., Handbuch der Software-Architektur, dpunkt.verlag, 2009
- Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996
- Stefan Roock and Martin Lippert. Refactorings in grossen Softwareprojekten. dpunkt-Verl., 1. au. edition, 2004.
- K. Schwaber and M. Beedle. Agile Software Development with Scrum. Pearson Studium, 2008
- Johannes Siedersleben. Moderne Softwarearchitektur. dpunkt.verlag, 2006
- Frank Simon, Olaf Seng, and Thomas Mohaupt. Code-Quality-Management. dpunkt.verlag, 2006
- Thomas Stahl and Markus Volter. Model-Driven Software Development Technology, Engineering, Management. Wiley, 2006

- Chris Wolf and Erick M. Halter. Virtualization: from the desktop to the enterprise. Apress; 1 edition, 2005
- Ian Sommerville. Software Engineering 8 (International Computer Science Series): Update. Addison Wesley; Ausgage: 8th ed., 2006.

7.2 Journals

- ACM Queue
- Elsevier Information and Software Technology
- Elsevier Journal of Systems and Software
- Elsevier Science of Computer Programming
- IEEE Computer
- IEEE Electronics Systems and Software
- IEEE Software
- IEEE Transactions on Software Engineering
- Springer Empirical Software Engineering
- Springer Lecture Notes in Computer Science
- Springer, The International Journal of Advanced Manufacturing Technology
- Wiley Journal of Software Maintenance and Evolution: Research and Practice (JSME)

7.3 Dissertations

- Cerulo L., On the Use of Process Trails to Understand Software Development (Analysis of historical data in combination with static and dynamic analysis) 2006
- Ciupke O., Problemidentifikation in objektorientierten Softwarestrukturen, The University of Karlsruhe, 2002
- Cubranic D., Project History as a Group Memory: Learning From the Past 2005
- Hassan A. E., Mining Software Repositories to Assist Developers and Support Managers, The University of Waterloo, 2004
- Moonen L., Exploring Software Systems, The University of Amsterdam, 2002
- Penta, M. D., Evolution Doctor: A Framework to Control the Evolution of Undocumented Software Systems 2003
- Robillard. Representing Concerns in Source Code. PhD thesis, 2003
- Wu J., Open Source Software Evolution and Its Dynamics, The University of Waterloo, 2006
- Zou Y., Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms, The University of Waterloo, 2003

7.4 Conferences and Workshops

- ACM SIGDOC '03,
- Agile Conference, 2007-2009. AGILE
- European Science Foundation Conference 2009
- Frontiers of Software Maintenance 2008. FoSM 2008.
- IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009. ECBS 2009.
- IEEE International Conference on Engineering Complex Computer Systems, 1998, 2000, 2005. ICECCS
- IEEE International Conference on Software Maintenance, 2003-2009, ICSM.
- IEEE International Symposium Software Metrics, 2005.
- IEEE International Workshop on Program Comprehension, 2004
- IEEE Software Engineering Workshop, 2008
- International Computer Software and Applications Conference, 2004
- International Conference on COTS-Based Software Systems, ICCBSS, 2006
- International Conference on Software Engineering, 2004. ICSE 2004.
- International Workshop on Principles of Software Evolution, 2003
- Joint Working IEEE/IFIP Conference on Software Architecture, 2009 and European Conference on Software Architecture. WICSA/ECSA 2009.
- OOPSLA'05
- Software Engineering Conference, 2004
- Working Conference on Reverse Engineering, 1999, 2000, 2005
- The Conference on The Future of Software Engineering, 2004
- The Working IEEE/IFIP Conference on Software Architecture, 2007. WICSA '07.
- Third ACM SIGSOFT Symposium on the Foundations of Software Engineering 1995

7.5 Interviews

- MDSD: Interview of MDSD researchers at FZI

7.6 Other

- AUTOSAR-Konsortium. Automotive open system architecture
- CRID 80321. Data mining survey, 2009
- Götzenauer, Master's thesis, Agile Methoden in der Softwareentwicklung: Vergleich und Evaluierung 2005

- Heise Developer SoftwareArchitekTOUR-Podcast, (http://www.heise.de/developer/podcast/itunes/heise-developer-podcast-softwarearchitektour.rss)
- MITRE, Center for Air Force C2 Systems, Bedford, Massachusetts, 1998
- Nokia corporation report. 2008
- SE Radio Podcast, (http://se-radio.net)
- Steffen Becker, Master's thesis, Carl von Ossietzky Universitaet Oldenburg, 2003
- Software Engineering Institute
- Technical report, The Federal Aviation Administration (FAA), 2000
- Technical report, Trinity College and Broadcom Eireann Research, Dublin, Ireland, 1997
- White Paper, George Mason University, 2005
- Versionone. State of agile survey, 2009

7.7 Search keywords

The following list comprises the most important search keywords which were used for the identification of documents and approaches which are relevant for this document (examples):

- agility and architecture
- software evolution
- strategies, strategy, tactic(s), method(s), approach
- software maintenance
- maintainability
- evolvability
- longevity
- modifiability
- flexibility
- sustainability
- COTS
- (data) mining
- virtualization
- software quality
- architecture compliance checking
- architecture analysis
- code and architecture consistency
- architecture(al) enforcements
- survey, evaluation

- taxonomy, classification
- empirical
- controlled experiment
- experience report
- And other

Chapter 8

Glossary

Agile methods are software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated. Agile methods generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals, (Source: http://en.wikipedia.org/wiki/Agile_software_development).

Antipatterns is a pattern that may be commonly used but is ineffective and/or counterproductive in practice. According to [BMMM98] there must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea: 1) Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and 2) a refactored solution exists that is clearly documented, proven in actual practice and repeatable.

Application migration is a transfer process of a control system on newer or some other version of the technical system (for the automation domain). Due to the domain specific migration process ("migration strategy") is usually plant or factory specific and is introduced gradually.

Architectural Description Methods According to ANSI/IEEE standard 1471-2000 a architecture description is a set of models (e.g., textual specifications or graphical diagrams (e.g. UML diagrams)), which document the software architecture.

Artifacts are tangible byproducts produced during the development of a software. Some artifacts (e.g., use cases, class diagrams, and other UML models, requirements and design documents) help describe the function, architecture, and design of software. Other artifacts are concerned with the process of development itself - such as project plans, business cases, and risk assessments, (Source: http://en.wikipedia.org/wiki/Artifact_%28software_development%29).

Automation domain is an appellation for automated plants, factories and utilities and their control systems. Such control systems usually contain real-time components and have client-server or multi-tier architectures with event-driven communication. They consist of distributed server nodes, client nodes, and embedded systems (e.g. controllers and field devices).

Automation Strategies are approaches, which automate software development, e.g., by increasing the abstraction level and generation of repetitive low-level code fragments.

Bad Smell or Code Smell is a symptom in the source code of a program that possibly indicates a deeper problem, [FBB+99].

COTS (Commercial off-the-shelf) is a term defining technology which is ready-made and available for sale, lease, or license to the general public. The term often refers to computer software or hardware systems and may also include free software with commercial support, (Source: http://en.wikipedia.org/wiki/Commercial_off-the-shelf).

Data Mining is the process of extracting patterns from data, (Source: http://en.wikipedia.org/wiki/Data_mining).

DCS (Distributed Control System) refers to a control system usually of a manufacturing system, process or any kind of dynamic system, in which the controller elements are not central in location but are distributed throughout the system with each sub-system controlled by one or more controllers. The entire system of controllers is connected by networks for communication and monitoring. (Source: http://en.wikipedia.org/wiki/Distributed_control_system).

Design pattern in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise. The idea was introduced by the architect Christopher Alexander, [Ale77], in the field of architecture and has been adapted for various other disciplines, including computer science, [GHJV95]. An organized collection of design patterns that relate to a particular field is called a pattern language. Original statement of Christopher Alexander: "The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Development environment consists of all development tools which are used for designing, implementing, and maintaining a software system. This covers editors, integrated development environments, debugging tools, testing tools, as well as versioning and bug tracking tools.

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, by means of various plug-ins, in other languages including C, C++, COBOL, Python, Perl, and PHP. During the last years Eclipse evolved from a pure development environment to a rich client platform, which can be used as a foundation for a wide range of applications.

Executable UML is a profile of the UML, that graphically specifies a system "at the next higher level of abstraction, abstracting away both specific programming languages and decisions about the organization of the software." The models are testable, and can be compiled into a less abstract programming language to target a specific implementation, (Source: http://en.wikipedia.org/wiki/Executable_UML).

Evolution see Software Evolution

Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge, [CE00].

IEC 61508 is a standard for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems provides general guidance on the selection of design techniques according to the safety criticality of a software element under design.

IEC 61850 is the international standard for communication networks and systems in substations, defines the communication between devices in the substation and related system requirements. It supports substation automation functions as well as their engineering.

Knowledge transfer is a process of passing knowledge within a company, department, or development team. Knowledge could range from domain-specific or application-specific knowledge to general knowledge like best practice or experience about past challenges and solutions. This can cover knowledge about system, processes, methodologies, techniques, tools, etc. Knowledge transfer can happen over time and space. A special challenge with represents to sustainability represents the knowledge transfer of design decisions, design rationale and selection evolution strategies.

Legacy system is a system, which is still valuable to its stakeholders despite of its degraded quality.

Lehman's Laws or so-called "Laws of Software Evolution" describe a set of behaviors (observations) in the evolution of proprietary software and are believed to apply mainly to monolithic, proprietary software. The laws predict that change is inevitable and not a consequence of bad programming and that there are limits to what a software evolution team can achieve in terms of safely implementing changes and new functionality. The laws we formulated by Lehman and Belady starting from 1972 during their research of evolution history of big software systems.

Longevity is the ability for a (software) system to be able stay alive for a long time (more than 10 years) and requires the ability to cope with a changing environment and changing user requirements during the whole period of life.

Long living systems are systems having a life cycle of more than 10 years. The life cycle ranges from first installation and deployment to final shutdown of the system.

Maintainability is, according to ISO/IEC 9126-1, the capability of the software system to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification.

Model Driven Architecture (MDA) is the standardization initiative of the Object Management Group (OMG) with respect to MDSD, (http://www.omg.org/mda/), and a flavor of the MDSD paradigm.

The core building blocks of MDA standard are UML2.0, Meta Object Facility (MOF), XML Metadata Interchange (XMI), the three kinds of models (PIM, PSM, PDM), Multi-Stage Transformations, Action Languages, the various core models, model marking, and Executable UML.

Model Driven Software Development (MDSD) is a software development paradigm based on modeling where models are considered equal to code, as their implementation is automated. Another common name is Model Driven Development, [SV06].

Modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.

OPC is open connectivity in industrial automation and the enterprise systems that support industry. Interoperability is assured through the creation and maintenance of open standards specifications.

Product lines see Software product line

Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some more form of forward engineering or restructuring.

Refactoring (noun) is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour of the software.

Refactor (verb) means to restructure software by applying a series of refactorings without changing the observable behaviour of the software.

Reference Architecture is an abstract software architecture, which defines structures and types of software elements, their allowed interactions and their responsibilities specific to an application domain. Structures are applicable for all systems within an application domain. They represent established basic constructions, that might contain collected experience of several engineering generations and are supported by a large community of researchers and practitioners. For example in compiler construction it is common to split components into lexical analysis (Scanner), syntactical analysis (Parser), semantic analysis, and generators, [RH09]. Other examples are Quasar, [Sie06], which is a reference architecture for business information systems and AUTOSAR, [AK], providing a reference architecture for software in automobiles. [RH09] distinguish three kinds of reference architectures: 1) functional, 2) logical, and 3) technical.

Software erosion is the decreasing quality of the internal structure of a software system, it may occur already at early development stages of the system.

Software evolution is a change process of a system (concerning both HW and SW) starting from its development and going on until system recycling, during which system changes into a different and usually more complex or better. System evolution is part of the system life cycle.

Software factory is a configuration of languages, patterns, frameworks, and tools that can be used to rapidly and cheaply produce an open-ended set of unique variants of a software product. It is not only intended for automating software development within an individual organization, but should promote the formation of software supply chains, [GS04].

Software migration is a change process during which system is being moved from one environment, technology or technique (meaning both, HW and SW) to another. This process is usually triggered either by change request or by own life cycle of environment, technology, or technique. Migration is a variant of reengineering in which the transformation is driven by a major technology change. In the automation domain an application migration is usually understood under the term migration.

Software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way, (http://www.sei.cmu.edu/architecture/start/glossary/).

Structural strategies provide ways of organizing the software structure,i.e., its architecture or design, in a way that dependencies are better controlled, changes are easier to make or at least are doable in a more systematic way.

Sustainability see Longevity.

Utility tree is a top-down vehicle for characterizing the "driving" attribute-specific requirements where nodes represent important quality goals and leaves represent scenarios, (http://www.sei.cmu.edu/architecture/start/glossary/).

Virtualization is a broad term which encompasses a number of different technologies. In the document's context virtualization is the "separation of a resource or request for a service from the underlying physical delivery of that service".

Bibliography

[80309] CRID 80321. Data mining survey, 2009.

[AABK10] Pekka Abrahamsson, Muhammad Ali Babar, and Philippe Kruchten. Agility and architecture: Can they coexist? *IEEE Software*, 27 , Issue:2:16–22, 2010.

[ACLM99] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in oo systems. *Proceedings. Sixth Working Conference on Reverse Engineering*, pages 136 – 144, 1999.

[AHGT06] B. Anda, K. Hansen, I. Gullesen, and H.K. Thorsen. Experiences from using a uml-based development method in a large organization. *Empirical Software Eng.*, vol. 11:555–581, 2006.

[AJaJR03] P. Abrahamsson, J.Warsta, and M. Siponen and. J. Ronkainen. New directions on agile methods: A comparative analysis. *25th International Conference on Software Engineering (ICSE'03)*, pages 244 – 254, 2003.

[AK] AUTOSAR-Konsortium. Automotive open system architecture, http://www.autosar.org.

[AL03] M. Alshayeb and Wei Li. An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on Software Engineering*, 29, Issue: 11:1043 – 1049, 2003.

[Ale77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, ISBN 0195019199, 1977.

[Amb02] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* Wiley, 2002.

[ASRW02] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods: Review and analysis. *VTT Publication*, 2002.

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

[Bab09] M. A. Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009.*, pages 81 – 90, 2009.

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik - Software-Entwicklung.* Spektrum-Akademischer Vlg, 2000.

[BBD00] Elizabeth Burd, Steven Bradley, and John Davey. Studying the process of software change: an analysis of software evolution. *Seventh Working Conference on Reverse Engineering, 2000. Proceedings.*, pages 232 – 239, 2000.

[Bec03] Stefen Becker. Cost model, decision support and selection process for cots. Master's thesis, Carl von Ossietzky Universität Oldenburg, 2003.

[Ben96] K Bennett. Software evolution: past, present and future. *Information and Software Technology*, 38(11):673–680, November 1996.

[BF07] Nord R. Bachmann F., Bass L. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, 2007.

[BI07] R. J. Barnett and B. V. Irwin. Virtualized systems and their performance: A literature review. 2007.

[BLBvV04] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (alma). *Journal of Systems and Software*, 69(1-2):129 – 147, 2004.

[BLR+97] J. Bisbal, D. Lawless, R. Richardson, D. O'Sullivan, B. Wu, J. Grimson, and V. Wade. A survey of research into legacy system migration. Technical report, Trinity College and Broadcom Éireann Research, Dublin, Ireland, 1997.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change:Â Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309, 2005.

[Bos00] Jan Bosch. *Design and Use of Software Architectures Adopting and evolving a product-line approach.* Addison-Wesley, 2000.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *International Conference on Software Engineering*, page 73, 2000.

[BR09] S. Brcina, R.and Bode and M. Riebisch. Optimisation process for maintaining evolvability during software evolution. *16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009. ECBS 2009.*, pages 196 – 205, 2009.

[BSB08] Christoph Bommer, Markus Spindler, and Volkert Barr. *Softwarewartung.* dpunkt.verlag, 2008.

[Bus01] Frank Buschmann. *A system of patterns.* Wiley, repr. edition, 2001.

[BZJ04] M.A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 309–318, 2004.

[CB10] James Coplien and Gertrud Bjørnvig. *Lean Architecture: for Agile Software Development.* John Wiley & Sons; Auflage: 1, 2010.

[CBBG02] Paul Clements, Felix Bachmann, Len Bass, and David Garlan. Documenting software architectures: Views and beyond (sei series in software engineering). 2002.

[CDP09] L. Canfora, G.and Cerulo and M. Di Penta. Tracking your changes: A language-independent approach. *Software, IEEE*, 26 , Issue:1:50 – 57, 2009.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[Cer06] Luigi Cerulo. *On the Use of Process Trails to Understand Software Development (Analysis of historical data in combination with static and dynamic analysis)*. PhD thesis, 2006.

[CF04] K. Conboy and B. Fitzgerald. Toward a conceptual framework of agile methods: A study of agility in different disciplines. *The 2004 ACM workshop on Interdisciplinary software engineering research*, pages 37 – 44, 2004.

[CH01] A. Cockburn and J. Highsmith. Agile software development: The people factor. *Computer*, pages 131–133, 2001.

[CHK+99] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, 2:3–30, 1999.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance: Research and Practice*, 13(1):3, 2001.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery:Â A Taxonomy. *IEEE Software*, 7(1):13, 1990.

[Ciu02] Oliver Ciupke. *Problemidentification in objektorientierten Softwarestrukturen*. PhD thesis, The University of Karlsruhe, 2002.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 – 493, jun 1994.

[CKK05] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures*. Addison-Wesley, 4. print. edition, 2005.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines Practices and Patterns*. Addison Wesley, 2002.

[Coc04] A. Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Longman, Amsterdam, 2004.

[CP07] Gerardo CanforaHarman and Massimiliano Di Penta. New Frontiers of Reverse Engineering. In *International Conference on Software Engineering*, pages 326–341, 2007.

[CPP10] I. Christou, S. Ponis, and E. Palaiologou. Using the agile unified process in banking. *Software, IEEE*, 27 Issue:3:72 – 79, 2010.

[CRR09] Lan Cao, B. Ramesh, and M. Rossi. Are domain-specific models easier to maintain than uml models? *Software, IEEE*, Volume: 26 , Issue: 4:19 – 21, 2009.

[CT98] Judith A. Clapp and Audrey E. Taub. A management guide to software maintenance in cots-based systems. *MITRE, Center for Air Force C2 Systems, Bedford, Massachusetts*, 1998.

[Cub05] Davor Cubranic. *Project History as a Group Memory: Learning From the Past*. PhD thesis, 2005.

[DAB08] W.J. Dzidek, E. Arisholm, and L.C. Briand. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *Software Engineering, IEEE Transactions on*, v 34 , Issue:3, 2008.

[DL00] R. Dumke and F. Lehner. *Software-Metriken Entwicklungen, Werkzeuge und Anwendungsverfahren*. Deutscher Unversitäts-Verlag, 2000.

[Dob02] E. Dobrica, L.; Niemela. A survey on software architecture analysis methods. *Transactions on Software Engineering*, 28(7):638–653, Jul 2002.

[DSB09] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Variability assessment in software product families. *Inf. Softw. Technol.*, 51(1):195–218, 2009.

[DSNB04] S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch. Cosvam: a technique for assessing software variability in software product families. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 458 – 462, 11-14 2004.

[EW09] SAAB Erik Wedin. Model-driven architecture and xtuml in practice. In *ESF Conference 2009, Seattle, USA*, 2009.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FCS+10] Davide Falessi, Giovanni Cantone, Salvatore Alessandro Sarcia, Guiseppe Calvaro, Paolo Subiaco, and Cristiana D'Amore. Peaceful coexistence: Agile developer perspectives on software architecture. *Software, IEEE*, 27 , Issue:2:23 – 25, 2010.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software:Â A Research Roadmap. In *International Conference on Software Engineering*, pages 37–54, 2007.

[Fra06] L. Francis. Lifetime procurement - look deep for dependability. *Electronics Systems and Software*, 4 , Issue:6:22 – 25, 2006.

[Gar00] David Garlan. Software architecture: a roadmap. *In Proceedings of the Conference on The Future of Software Engineering*, pages 91 – 101, 2000.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, pages 207–236, 2006.

[GG08] Michael W. Godfrey and Daniel M. German. The Past, Present, and Future of Software Evolution. In *Proc. 24th Int. Conf. on Software Maintenance*, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[Gia05] Piergiorgio Di Giacomo. Cots and open source software components: Are they really different on the battlefield? *Lecture Notes in Computer Science, COTS-Based Software Systems*, 3412/2005:301–310, 2005.

[GNW09] Erich Gamma, Lee Nackmann, and John Wiegand. *eclipse Modeling Project A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[Got09] G. Goth. Agile tool market growing with the philosophy. *Software, IEEE*, 26 Issue:2:88 – 91, 2009.

[GR05] E. Germain and P. Robillard. Engineering-based processes and agile methodologies for software development: a comparative case study. *The Journal of Systems and Software, Elsevier*, pages 17 – 27, 2005.

[GS04] Jack Greenfield and Keith Short. *Software Factories Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[Göt05] Götzenauer. Agile methoden in der softwareentwicklung: Vergleich und evaluierung. Master's thesis, 2005.

[Has06] Ahmed E. Hassan. Mining software repositories to assist developers and support managers. In *Proceedings of ICSM 2006: IEEE International Conference on Software Maintenance, Chicago, Philadelphia, USA*, pages pp. 339–342, Sept. 24-27, 2006.

[Has08] A.E. Hassan. The road ahead for mining software repositories. *Frontiers of Software Maintenance, 2008. FoSM 2008*, pages 48 – 57, 2008.

[HG05] Elisabeth Hansson and Göran V. Grahn. One global cots-based system to replace 20+ local legacy systems. *Lecture Notes in Computer Science, COTS-Based Software Systems*, 3412/2005:144–145, 2005.

[HH04] A.E. Hassan and R.C. Holt. Using development history sticky notes to understand software architecture. *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 183 – 192, 2004.

[HJH05] A.E. Hassan, Zhen Ming Jiang, and R.C. Holt. Source versus object code extraction for recovering software architecture. *Reverse Engineering, 12th Working Conference on*, page 10, 2005.

[HLP05] Jesper Holck, Michael Holm Larsen, and Mogens Kühn Pedersen. Managerial and technical barriers to the adoption of open source software. *Lecture Notes in Computer Science, COTS-Based Software Systems*, 3412/2005:289–300, 2005.

[HRJ+04] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The dublo architecture pattern for smooth migration of business information systems: an experience report. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 117 – 126, 23-28 2004.

[HS96] Brian Henderson-Sellers. *Object-oriented Metrics Measures of Complexity.* Prentice Hall, 1996.

[HSS09] Carl Hinsman, Neeraj Sangal, and Judith A. Stafford. Achieving agility through architecture visibility. *QoSA, Lecture Notes in Computer Science, Springer*, Vol. 5581:116–129, 2009.

[HYCM09] G.K. Hanssen, A.F. Yamashita, R. Conradi, and L. Moonen. Maintenance and agile development: Challenges, opportunities and future directions. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 487 – 490, 2009.

[IF10] Ayelet Israeli and Dror G. Feitelson. The linux kernel as a case study in software evolution. *Journal of Systems and Software*, Volume 83 , Issue 3:485–501, 2010.

[Ish08] Mark Isham. Agie architecture is possible - you first have to believe! *Conference Agile, 2008. AGILE '08.*, pages 484 – 489, 2008.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering (2nd ed.).* Addison-Wesley Longman, 2002.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[KH05] Gerald Kotonya and John Hutchinson. Analysing the impact of change in cots-based systems. *Lecture Notes in Computer Science, COTS-Based Software Systems*, 3412/2005:212–222, 2005.

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on*, pages 68 –78, 10-14 1998.

[KMR08] J. Knodel, D. Muthig, and D. Rost. Constructive architecture compliance checking — an experiment on support by live feedback. *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 287 – 296, 2008.

[KMSN+06] G.A. Kajko-Mattsson, M.and Lewis, D. Siracusa, T. Nelson, N. Chapin, M. Heydt, J. Nocks, and H.; Snee. Long-term life cycle impact of agile methodologies. *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 422 – 425, 2006.

[Koe09] Koehnemann. Experiences applying agile practices to large systems. *Agile Conference, 2009. AGILE '09*, pages 295 – 300, 2009.

[Kos03] J. Koskela. Software configuration management in agile methods. *VTT Publication*, 2003.

[KP07] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, page 12, 2007.

[Kru07] Phillipe Kruchten. Voyage in the agile memeplex. *Queue*, Volume 5 , Issue 5:1, 2007.

[LA04] P. Lappo and H. C. T. Andrew. Assessing agility. *Extreme Programming and Agile Processes in Software Engineering, Lecture Notes in Computer Science*, Volume 3092/2004:331 – 338, 2004.

[Laa08] Laanti. Implementing program model with agile principles in a large software development organization, nokia corporation. 2008.

[LBCC08] Rikard Land, Laurens Blankers, Michel Chaudron, and Ivica Crnkovic. Cots selection best practices in literature and in industry. *Lecture Notes in Computer Science, High Confidence Software Reuse in Large Systems,*, pages 100–111, 2008.

[LBCK06] Jingyue Li, Finn Olav Bjørnson, Reidar Conradi, and Vigdis B. Kampenes. An empirical study of variations in cots-based software development processes in the norwegian it industry. *Empirical Software Engineering*, Volume 11, Number 3:433–461, 2006.

[LCB+09] Jingyue Li, R. Conradi, C. Bunse, M. Torchiano, O. Slyngstad, and M. Morisio. Development with off-the-shelf components: 10 facts. *Software, IEEE*, 26 Issue:2:80 – 87, 2009.

[LCS+05] Jingyue Li, R. Conradi, O.P.N. Slyngstad, C. Bunse andU. Khan, M. Torchiano, and M. Morisio. Validation of new theses on off-the-shelf component based development. *Software Metrics, 2005. 11th IEEE International Symposium*, pages 26 – 26, 2005.

[Lew00] Patrick Lewis. Lessons learned in developing commercial off-the-shelf (cots) intensive software systems. Technical report, The Federal Aviation Administration (FAA),, 2000.

[LL02] Karl Leung and Hareton Leung. On the efficiency of domain-based cots product selection method. *Information and Software Technology*, Volume 44, Issue 12:703–715, 2002.

[LL06] Jochen Ludewig and Horst Lichter. *Software Engineering.* Dpunkt Verlag, 2006.

[LMM01] Howard F. Lipson, Nancy R. Mead, and Andrew P. Moore. Can we ever build survivable systems from cots components? *Lecture Notes in Computer Science, Advanced Information Systems Engineering*, 2348/2006:216–229, 2001.

[LR00] M. M. Lehman and J. F. Ramil. Software evolution in the age of component-based software engineering. *IEE Proceedings - Software*, 147:249–255, 2000.

[LST+08] Jingyue Li, O.P.N. Slyngstad, M. Torchiano, M. Morisio, and C. Bunse. A state-of-the-practice survey of risk management in development with off-the-shelf software components. *Software Engineering, IEEE Transactions on*, 34 , Issue:2:271 – 286, 2008.

[Mad10] James Madison. Agile architecture interactions. *Software, IEEE*, 27 , Issue:2:41 – 48, 2010.

[Mar09] Robert C. Martin. *Clean Code A Handbook of Agile Software Craftmanship.* Prentice Hall, 2009.

[MB02] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Jacoboson, Ivar.

[MC07] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, Volume 12, Number 5:471–516, 2007.

[MD01] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *International Conference on Software Engineering*, page 83, 2001.

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449, 2002.

[Men05] Daniel A. Menascé. Virtualization: Concepts, applications, and performance modeling. *White Paper, George Mason University*, 2005.

[Men08] Tom Mens. *Software Evolution.* Springer Berlin Heidelberg, 2008.

[MFRD08] T. Mens, J. Fernandez-Ramil, and S. Degrandsart. The evolution of eclipse. *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.*, pages 386 – 395, 2008.

[MH08] H. Malik and A.E. Hassan. Supporting software evolution using adaptive change propagation heuristics. *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.*, pages 177 – 186, 2008.

[MNS95] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models 1995. acm. *In Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, page 18–28, 1995.

[Moo02] Leon Moonen. *Exploring Software Systems.* PhD thesis, The University of Amsterdam, 2002.

[MRS10] K. Mohan, B. Ramesh, and V. Sugumaran. Integrating software product line engineering and agile development. *Software, IEEE*, 27 Issue:3:48 – 55, 2010.

[MT04] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126, 2004.

[MT05] Piyush Maheshwari and Albert Teoh. Supporting atam with a collaborative web-based software architecture evaluation tool. *Science of Computer Programming*, 57(1):109–128, 2005.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in Software Evolution. In *IWPSE*, page 13, 2005.

[Nus01] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34 , Issue:3:115 – 119, 2001.

[OMB03] C. OReilly, P. Morrow, and D. Bustard. Lightweight prevention of architectural erosion. *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 59 – 64, 2003.

[OMG03] OMG. MDA Guide Version 1.0.1. *http://www.omg.org/cgi-bin/doc?omg/03-06-01*, 2003.

[Pen03] Massimiliano Di Penta. *Evolution Doctor: A Framework to Control the Evolution of Undocumented Software Systems.* PhD thesis, 2003.

[PFP02] S. R. Palmer, M. Felsing, and S. Palmer. *A Practical Guide to Feature-Driven Development.* Prentice Hall International, 2002.

[Pos09] D. Poshyvanyk. Using information retrieval to support software maintenance tasks. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 453 – 456, 2009.

[PP05] M. Pikkarainen and U. Passoja. An approach for assessing suitability of agile solutions: A case study. *The 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005)*, pages 171 – 179, 2005.

[PTD+09] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendon. Static architecture conformance checking - an illustrative overview. *Software, IEEE*, PP , Issue:99, 2009.

[QHS06] A. Qumer and B. Hendersson-Sellers. Comparative evaluation of xp and scrum using the 4d analytical tool (4-dat). *The European and Mediterranean Conference on Information Systems (EMCIS)*, 2006.

[RBB+03] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, , and Betsy Clark. Eight lessons learned during cots-based systems maintenance. *IEEE Software*, 20:94–96, 2003.

[RBBC04] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark. Cots-based systems – twelve lessons learned about maintenance. *Lecture Notes in Computer Science, COTS-Based Software Systems*, 2959/2004:137–145, 2004.

[RH09] Ralf Reussner and Wilhelm Hasselbring. *Handbuch der Software-Architektur.* dpunkt.verlag, 2. edition, 2009.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[RL04] Stefan Roock and Martin Lippert. *Refactorings in großen Softwareprojekten*. dpunkt-Verl., 1. aufl. edition, 2004.

[Rob03] Robillard. *Representing Concerns in Source Code*. PhD thesis, 2003.

[RRS09] Francisco Javier Romero Rojo, Rajkumar Roy, and Essam Shehab. Obsolescence management for long-life contracts: state of the art and future trends. *The International Journal of Advanced Manufacturing Technology*, 2009.

[SA08] O. Salo and P. Abrahamsson. Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum. *Software, IET*, 2 , Issue:1:58 – 64, 2008.

[SB03] H.M. Sneed and P. Brossler. Critical success factors in software maintenance: a case study. *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 190 – 198, 2003.

[SB08] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Pearson Studium, 2008.

[SD07] K. Silva and C. Doss. The growth of an agile coach community at a fortune 200 company. *AGILE 2007*, pages 225 – 228, 2007.

[SD08] Marco Sinnema and Sybren Deelstra. Industrial validation of covamof. *J. Syst. Softw.*, 81(4):584–600, 2008.

[Sie06] Johannes Siedersleben. *Moderne Softwarearchitektur*. dpunkt.verlag, 2006.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Som06] Ian Sommerville. *Software Engineering 8 (International Computer Science Series): Update*. Addison Wesley; Auflage: 8th ed., 2006.

[SSM06] Frank Simon, Olaf Seng, and Thomas Mohaupt. *Code-Quality-Management*. dpunkt.verlag, 2006.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development Technology, Engineering, Management*. Wiley, 2006.

[Tar09] A. Tarvo. Mining software history to improve software maintenance quality: A case study. *Software, IEEE*, Volume: 26 , Issue: 1:34 – 40, 2009.

[TFR02] D. Turk, R. France, and B. Rumpe. Limitations of agile software processes. *The 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2002)*, pages 43 – 46, 2002.

[TFR05] D. Turk, R. France, and B. Rumpe. Assumptions underlying agile software-development processes. *Journal of Database Management (JDM)*, Volume 16, issue 4:62 – 87, 2005.

[TH03] S. Tilley and S. Huang. A qualitative assessment of the efficacy of uml diagrams as a form of graphical documentation in aiding program understanding. *Proc. ACM SIGDOC '03*, pages 184–191, 2003.

[THV09] Antony Tang, Jun Han, and Rajesh Vasa. Software architecture design reasoning: A case for improved methodology support. *Software, IEEE*, 26 , Issue:2:43–49, 2009.

[TR08] Masoumeh Taromirad and Raman Ramsin. Cefam: Comprehensive evaluation framework for agile methodologies. *32nd Annual IEEE Software Engineering Workshop*, pages 195–204, 2008.

[TRDL07] M. Torchiano, F. Ricca, and A. De Lucia. Empirical studies in software maintenance and evolution. *IEEE International Conference on Software Maintenance, 2007. ICSM 2007.*, pages 491 – 494, 2007.

[Try97] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Eng.*, vol. 2:201–207, 1997.

[TTBS07] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. Empirical studies in reverse engineering:Â state of the art and future trends. *Empirical Software Engineering*, 12(5):551, 2007.

[Van07] Arie Van Deursen. Model-driven software evolution: A research agenda . In *Proc. Int'l Workshop on Model-Driven Software Evolution*, June 2007.

[VD00] Mark R. Vigder and John Dean. Maintaining cots-based systems. *Fifth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'06)*, pages 11–18, 2000.

[vdMRSJ05] Meine van der Meulen, Steve Riddle, Lorenzo Strigini, and Nigel Jefferson. Protective wrapping of off-the-shelf components. *Lecture Notes in Computer Science, COTS-Based Software Systems*, 3412/2005:168–177, 2005.

[Ver09] Versionone. State of agile survey, 2009.

[Vor07] U. Vora. Architectural design methodologies for complex evolving systems. *12th IEEE International Conference on Engineering Complex Computer Systems, 2007.*, pages 197 – 206, 2007.

[WH05] Chris Wolf and Erick M. Halter. *Virtualization: from the desktop to the enterprise.* Apress; 1 edition, 2005.

[WK04] Weihang Wu and Tim Kelly. Safety tactics for software architecture design. *Proceedings of the 28th Annual International Computer Software and Applications Conference*, 01:368–375, 2004.

[WKLA04] L. Williams, W. Kerbs, L. Layman, and A. Anton. Toward a framework for evaluating extreme programming. *The 8th International Conference on Empirical Assessment in Software Engineering (EASE 04)*, pages 11 – 20, 2004.

[WSV05] Lei Wu, H. Sahraoui, and P. Valtchev. Coping with legacy system migration complexity. *10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings.*, pages 600 – 609, 2005.

[Wu06] Jingwei Wu. *Open Source Software Evolution and Its Dynamics.* PhD thesis, The University of Waterloo, 2006.

[WYL08] M. Wermelinger, Yijun Yu, and A. Lozano. Design principles in architectural evolution: A case study. *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.*, pages 396 – 405, 2008.

[XCI09] Guowu Xie, Jianbo Chen, and Neamtiu I. Towards a better understanding of software evolution: An empirical study on open source software. *IEEE International Conference on Software Maintenance, 2009. ICSM 2009.*, 51 - 60, 2009.

[YHS09] T. Yoshikawa, S. Hayashi, and M. Saeki. Recovering traceability links between a simple natural language sentence and source code using domain ontologies. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 551 – 554, 2009.

[Zou03] Ying Zou. *Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms*. PhD thesis, The University of Waterloo, 2003.