T U M

I N S T I T U T F Ü R I N F O R M A T I K

Selected Topics in Software Quality

Stefan Wagner, Florian Deissenboeck, Benjamin Hummel,
Elmar Juergens, Benedikt Mas y Parareda, Bernhard Schaetz
(Eds.)

TUM-I0824
Juli 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

Preliminaries

Software quality is a make-or-break criterion for acceptance and success of software systems. Therefore, developers have to pay great attention to performance, stability and the long-term cost efficiency of software systems. Key to achieve these attributes is a high software quality. However, even the term "software quality" itself is highly disputed and no broadly accepted definition exists. At the same time, it is clear that achieving this elusive goal is one of the greatest challenges in software enigneering and that due to its multi-faceted nature, all activities exercised as part of a development effort need to contribute to achieve high quality software.

This report offers insight into selected aspects in the field of software quality. All of the topics are active fields of research and innovative ideas and results are shown. The report is a result of the master seminar "Software Quality" at the Technische Universität München. To ensure practical relevance, the seminar was carried out in collaboration with itestra GmbH.

July 2008

Stefan Wagner, Florian Deisenboeck, Benjamin Hummel,
Elmar Juergens, Benedikt Mas y Parareda, Bernhard Schaetz
Supervisors of the seminar

Table of Contents

Quality Requirements

Christian Viezens

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
viezens@in.tum.de

Abstract. To assure quality of a software product the fundamental step in the engineering processes is the elicitation and specification of quality requirements that the product must meet. Correct and complete elicited and clearly formalized quality requirements are the base of the product's quality verification. Furthermore, the specification of quality requirements saves time and cost and is much more efficient then adjusting the characteristics of software product features during design, implementation or further phases of the product life cycle. The later in the software development process missing or conflicting quality requirements are detected the more expensive it will be to fix it. This paper focuses on the need of completely elicited, documented and measurable quality requirements. As quality models are a widely applied way of defining quality, a short introduction in those models is shown and the spectrum of practical quality requirements engineering approaches discussed in the literature for finding, eliciting and formalizing quality requirements is highlighted. Subsequently, four of the most interesting and cited practical approaches are presented in more detail. The characteristics and differences of those four approaches are discussed and finally concluded.

1 Introduction

Quality is an essential need in high critical software systems like embedded flight control systems. Furthermore, it is a key differentiator between a software product and its competitor products. However, it is a very hard challenge to assure quality in todays software systems because on the one hand software system's size, lifetime and complexity is continuously growing and on the other hand there is often not much flexibility to deadlines and budget. This implies a need for reliable, systematical and methodical software engineering concerning quality. Since quality is very complex and very hard to achieve this is big challenge. One possibility to meet this challenge is to define quality models which can be used as a common terminology for quality. Based on these models a subsequent step in the engineering process is to specify the quality requirements the products must meet. Without a well-defined set of quality requirements, software projects are vulnerable to failure [1]. Using quality requirements in software engineering gives a common terminology for software and requirement engineers to talk with stakeholders like users about the desired quality aspects of the product from the outset. As the quality requirements in sum are the specific product quality that has to be reached, they are a mighty tool for quality measurement and assurance. Once they are elicited and clearly specified, they can be measured during the whole development process. However, the are the following questions. What exactly are quality requirements? Are they comparable to functional requirements (FR) or non-functional requirements (NFR) and how can a requirement engineer identify and elicit them? Is there a practical and standardized way to specify them so that they are measurable?

1.1 Problem

The fundamental problem is that there is no clear definition of quality requirements in the literature. Some people understand them as a special part of NFR, some see them as equal to NFR but others use their own definitions. Customers call them generic quality attributes and software engineers call it NFR [2]. Another challenge dealing with quality requirements is that they are hard to elicit. This aspect depends on the nature of quality, it is complex and hard to define. So the question is how to capture an essential set of quality requirements for a software systems. This set has to be complete, no requirements should be missed, and it should be rather minimal than sophisticated and crammed with redundant or mutually conflicting requirements. The certain way to find all essential requirements is to introduce the end-users and other stakeholders. However, what is a practical method to bring to light what quality attributes the different stakeholders like developers, maintenance stuff or the end-user desire? Many of the problems in software development had their root cause in insufficient understanding of the customer and unclear requirements [3]. The proximate challenge, once a set of quality requirements is found, is to refine them that they become unambiguous and further, they have to be formalized so that they are traceable and measurable. Yet, the past shows that it is very hard to find criteria and metrics which can be applied to them.

1.2 Contribution

This paper gives a short literature review about approaches dealing with eliciting and documenting quality requirements. After defining quality requirements in Section 2 quality models are discussed and an example model, the ISO/IEC 9126 is presented in Section 3. Subsequently, an overview of the software engineering approaches for eliciting and documenting quality requirements is shown in Section 4 and four of the most relevant and cited are explained in more detail and illustrated by practical applications. The assets, drawbacks and differences between the approaches are discussed in Section 5 and finally, in Section 6, a conclusion is presented.

2 Definition of Quality Requirements

There is no clear definition of quality requirements in the literature due to the fact that there are still a lot of research activities focusing on this challenge. In requirements engineering, software and systems requirements are classified in functional requirements and non-functional requirements. Functional requirements are like verbs and non-functional requirements are like adjectives or characteristics [4]. This indicates that quality requirements can be associated with non-functional requirements. Quality requirements are usually seen as part of the non-functional requirements of a system [5]. This paper use the latter definition, obviously not all non-functional requirements are quality requirements, for example legal and political issues, or cost of ownership, but all quality requirements are NFR because they describe characteristics.

3 Quality Models

To talk about quality requirements a common terminology is needed which gives a precise definition of quality. [6] categorize approaches to define quality in guidelines, metric-based approaches, quality modeling and processes and process models. The quality requirements approaches presented in this paper are based on quality models.

3.1 Defining Quality by Structuring

Quality models give abstract definitions of the most important quality attributes (QA). They try to categorize quality in these quality attributes or quality characteristics which typical depend on a certain domain, project or organization context. Typically, QA are structured and broken down to sub attributes. A fine structured model leads to a level of sub attributes which are not further separable which means that they are very well and clearly specified. Such a clear specification of a sub attribute make them measurable so a metric can be applied. Hence, quality models are a good tool to define quality at any level of granularity which means that they are a good base for quality requirements engineering. As an example approach of a quality model this paper illustrates the quality model of the International Standard Organization (ISO) below. Due to the fact of limited space in this paper quality models can not be discussed in more detail but it is to say that [6] points out quality modeling has been and continues to be a very active field of research. Existing quality models lack in accessibility, justification, homogeneity and operationalization. Another lack in existing quality models is that there is usually no explicit modeling for stakeholder activities which concern quality attributes although activities are the biggest costs factor [7].

3.2 ISO/IEC 9126

The ISO/IEC 9126 [8] is a well defined international standard model, so it as a good base for conversation and is an ideally example. It is concerned primarily with the definition of six high level quality characteristics and attributes respectively for software products. The main characteristics are broken down in further sub-characteristics which can be refined to a desired level of abstraction, and additionally the ISO gives suggestion and ideas for refinements of these characteristics. An overview of the high level attributes is given below.

1. **Functionality** describes the existence, qualification and effects of the required functions of the software system using sub-characteristics like *suitability*, *accuracy* or *security*.
2. **Reliability** describes the capability of the software to maintain its level of performance under stated conditions for a stated period of time. For example, the amount of failures and the ability to recover system and data after failure with regard to time and costs can be broken down in *maturity* and *recoverability* respectively.
3. **Usability** is characterized by the effort needed for use and the individual assessment of such use, by a stated or implied set of users. Further sub categorization could lead to *learnability* which is users effort to learn the handling of the software application or *operability* which is users effort to handle the software application.
4. **Efficiency** is the relationship between the level of performance of the software and the amount of resources used, under stated conditions. *Time* and *resource behaviour* and also *compliance* are suggested sub attributes.
5. **Maintainability** characterize the effort needed to make specified modifications to the software. Modification can cover fixes, advancements or adaption of the environment, the requirements or of the specification. Sub attributes can be *stability analyzability*, *changeability* and so on.
6. **Portability** is the attribute which refers to the ease of software to be transferred from one environment (hardware, software or organizational environment) to another which can be described in sub attributes like *installability* or *replaceability*.

3.3 Discussion

As an typical example of a quality model the biggest advantage of the ISO/IEC 9126 is that it is an international standard which is accessible worldwide and which provides a common terminology for requirements engineers. It covers quality requirements and NFR respectively with high-level preciseness and proposes a structured decomposition of quality. It can be used as a initial base model but it has to be refined because the course-grained decomposition is not suitable for an actual assessment. So an individual refinement depending on project and system context and breaking down the quality attributes in measurable criteria has to be down when applying. Additionally, the model fail to indicate how to design and trace the quality attributes for a specific project or domain context. All attributes are covered equal, there is no distinction between use cases or activities like development, maintenance or usage of the system. Furthermore, there is no suggestion how to prioritize the quality requirements or how to detect and mark conflicts between them. Finally, the ISO/IEC 9126 provides common terminology and serves as a solid base which can be used as an initial quality model in software engineering but projects experiences should be used to refine the model down to a metric level for specific domains and contexts. Besides, it should be considered that features like activity modeling and conflict detection are not provided by the model.

4 Approaches for Quality Requirements

Lots of related work and research activities are concerned with quality, non-functional requirements or just one special quality attribute, e.g. usability, maintenance or security. Even so only few approaches are currently being applied in industry and none of them is really at a level of usage for education and apprenticeship at universities or software engineering schools. However, one of the first comprehensive methods for specifying NFR is described in [9]. This framework, provides detailed guidance on NFR refinement and includes identification and modeling of relationships between them. As the focus is on refinement, no detailed elicitation support or guidance for documentation is given. In addition, it provides no assistance in relating NFR with FR and architecture. [10] shows an approach on how to combine non-functional requirements and use cases which are elicited separately and are then combined to make sure that the use cases satisfy the non-functional requirements. Boehm and In introduce in [1] an approach to identify quality requirements conflicts. The main focus is on conflict identification using a knowledge-based tool that helps stakeholders to analyze requirements and identify conflicts among them, rather then elicitation and specification. Another approach to classify quality requirements based on cost is proposed in [11]. There the quality requirements are directly derived from the business goals to determine the costs of each requirement. A method of capturing, structuring and also verifying of quality requirements using activity-based quality models is discussed in [7]. The idea of that approach is that quality requirements are classified by stakeholder activities on quality attributes. For example, usability has the activity "usage" by the stakeholder "end-user" and the activity "training" for the stakeholder "teachers". Further they introduce prioritization of the quality attributes and conflict resolution of concurrent quality attributes. A comprehensive survey on NFR approaches can be found in [12].

This paper illustrates in the following section four approaches concerning quality requirements engineering. The approaches are chosen because they have relatively many citations in literature and all of them concerning the problems of elicitation and specification of quality requirements described above by suggesting detailed solutions. While the first three approaches, the experience-based NFR method by Doerr et al., the approach of dealing with NFR proposed by Ebert, and the Volere Requirements Process by Robertson and Robertson, concern non-functional requirements in general, the last one focus in the treatment of the special quality characteristic dependability.

4.1 Approach by Doerr et al.

The experience-based NFR method by Doerr et al. is introduced in [13]. The objective of this method is is to achieve an essential and sufficient set of measurable and traceable non-functional requirements. It gives complete guidance for the process of requirements elicitation and documentation so the main features of the method are:

- a process for common treatment of the high level quality attributes,
- experience-based quality models that capture experience with general characteristics of quality attributes, metrics to measure and means to achieve them,
- distinction between different types of quality attributes,
- detailed elicitation guidance in terms of checklists and a prioritization questionnaire,
- documentation guidance by providing document structure and template,
- use of rationales to justify non-functional requirements,
- treatment of non-functional requirements together with functional requirements and system architecture, and
- requirements management support including dependencies analysis.

The method distinguishes between quality attributes and non-functional requirements. A quality attribute is a non-functional characteristic of a system, user task, system task or organization and is captured in a quality model. Non-functional requirements are captured in documents based on templates and describe a certain value of a quality attribute that should be achieved in a specific project and associate it with a metric. For example, a quality model for the high-level quality attribute *Efficiency* can have a sub level quality attribute *Resource Utilisation* and one of the corresponding non-functional requirements for that quality attribute could be *Capacity*. A notation based on goal graphs for dependencies and refinement is used, so the relationship between quality attributes such as *Efficiency* and *Maintainability* can be designed by that notation in the reference quality model.

The procedure and structure of the method is illustrated in Fig. 1. In the first step high-level quality attributes are prioritized using a questionnaire and quality models are selected. After selecting the quality attributes the reference quality models are tailored by domain experts in a workshop and dependencies will be identified additionally. Based on the tailored quality models the reference checklists and templates will be tailored for the specific needs. In a second workshop the tailored templates, the tailored checklists and the initial functional requirements will be used to elicit and document the non-functional requirements. Since the method is experienced-based the artifacts (questionnaire, reference model, reference checklists, and reference template) are initially developed from literature but evolve from project to project which will result in more mature quality models.

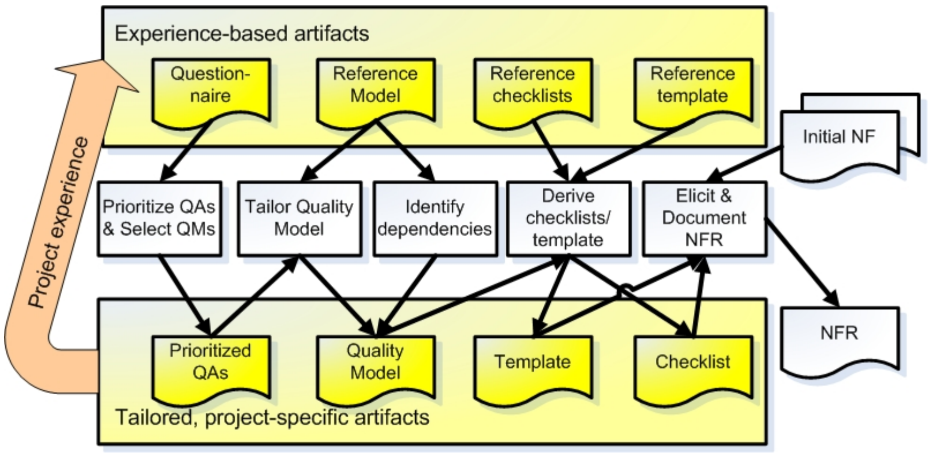


Fig. 1. Overview on Doerr's approach (adapted from [13])

Case studies The method was applied in three case studies. A wireless plant control project, a multi-functional printer system and a geographical information system. The first case study was used for non-functional requirements elicitation in a development project of a wireless plant control system. This system is a distributed and embedded system. The focus was on *Reliability* due to loss of production, *Efficiency* due to the needs that the application must run on hand-held devices, and *Maintainability* due to further extensions and maintenance. Because the case study was the first time the method was used, there was no previous experience which implied that the initial models were based directly on the ISO/IEC 9126. The initial set of artifacts were requirements documents and a non systematic set of non-functional requirements, so the first step was to transform all initial requirements in use cases. In two subsequent workshops with stakeholders the QM of maintainability, reliability, and efficiency were tailored and based in that tailoring the NFR were elicited. It was observed that the most time was spent for the elicitation of maintainability and gathered experiences showed that the prioritization of the QA was helpful. At the end new non-functional requirements were found in an efficient way and the three quality models of the corresponding quality attributes were enriched. Only 5 of 54 NFR were not measurable which means that 90 percent of the elicited NFR are measurable and a very important advantage was the early recognition of conflicts between requirements by performing a dependency analysis based on the goal graphs.

The second case study was performed for a multi-functional printer system project which is mainly based on integrated office equipment and embedded systems. The focus was on just one single quality attribute, *Efficiency*. Again, two workshops were performed to tailor the quality models for efficiency and to elicit the non-functional-requirements. At the end 16 new NFR were found. A web based geographical information system project was subject to the last case study where security was prioritized. All together the authors conclude that the experienced-based NFR method needs more effort then before but brings a positive return of investment due to reuse of models, completeness and correctness of NFR. The case studies also showed that the method gave sufficient guidance in eliciting and documenting NFR. Actually, the reference QM were helpful by capturing experience and offering means of communication. No irrelevant NFR were elicited, in fact the prioritization of the quality attributes and the tailoring were helpful to find an essential set of NFR which consists of measurable and traceable elements.

4.2 Approach by Ebert

Ebert discusses in [2] his approach to specify, trace and measure NFR and illustrates practical guidelines. Furthermore he provides prioritization of the elicited non-functional requirements using tables and suggests the design of roadmaps to achieve them. Ebert does not use the quality model ISO/IEC 9126, he introduces his own experience based quality attributes which are similar to ISOs but reasoned them not in detail. Actually, he distinguishes between user-oriented NFR and development-oriented NFR.

User-oriented NFR:

- performance (e.g., call set up time),
- reliability (e.g., critical failures per month),
- availability (e.g., supplier dependent total downtime per year),
- failure tolerance (e.g., failure recovery upon restart)
- usability (e.g., reaction time upon performing changes of ISDN features),
- correctness (e.g., dropped cell rate per second).

Development-oriented NFR:

- extendibility (e.g., free memory space of modules);
- maintainability (e.g., few ripple effects of changes);
- readability (e.g., consistency of code and comments);
- reusability (e.g., clearly defined and separated functionality of distinct components);
- fault tolerance (e.g., distributed functionality);

Guidelines for specifying and tracing NFR The first step of this iterating process is to elicit the user-oriented NFR according to contracts, proposals, required standards and marketing strategies. Separated from them the development-oriented NFR according to internal product or platform strategies can be collected in a further step. After that a technical analysis of the collected non-functional requirements in terms of impact, effort and interactions is proposed to performed. Based on the technical analysis a commercial analysis of the non-functional requirements is intended. By assigning priorities to the non-functional requirements a conflict resolution is aided. Those NFR with highest priority are selected and a roadmap which shows how to achieve them can be created. Specific measures for in-process quality checks and risk assessment will also be defined. Furthermore, an establishment of continuous risk assessment for each milestone of the NFR roadmap and of the project's master plan is advised and a design scenario for covering these NFR is suggested to be built. Due to the broad relationship with other requirements, Ebert mentions that this should start already during architectural design. Actually, a very important step is to measure and verify intermediate goals that have to be achieved according to the NFR roadmap.

Conflict resolution for interacting NFR The next step is to analyze conflicts between non-functional requirements. For example, performance might interfere with maintainability because complex coded algorithms are indeed high-performance but hard to understand for maintenance personal. Thus, it is necessary to prioritize such conflicting non-functional requirements after having them identified.

Guidelines for measuring NFR Measurement of non-functional requirements is primarily achieved by reviews because of the lack of clearly defined metrics. This includes the plan and effort estimation on the base of allocated requirements. It has to be noted that Ebert states that the achievement of milestones and related work products should be also periodically reviewed. He propose to use checklists for measuring NFR. This lists should contain properties of the software that together determine whether or how far the criteria have been met. A goal oriented approach is to find out goals and break these goals down according to the specific environment to find sub goals. Questions related to achieving these goals and finally metrics have to be selected that clearly answer the questions. These metrics can be numbers with units or on nominal scale.

4.3 Approach by Robertson and Robertson

In [4] an extensive requirements engineering approach called *Volere Requirements Process* is presented. It contains a set of requirements engineering templates designed by Suzanne and James Robertson which are described in detail in their book, this set is called Volere. The approach points out how to elicit and define requirements using Volere. This paper will concentrate on that part of the process which deals with NFR. This part consist of the sub-processes *Trawl for Requirements*, *Prototype the Requirements*, *Write the Requirements*, and *Quality Gateway* which are introduced below. Fig. 2 gives an overview of the processes.

The Volere Requirements Process The process is an iterative generic method for requirements specifications. One part of the process is about discovering, eliciting, specifying and testing non-functional requirements, i.e. quality requirements. The iteration contains the extraction of the potential requirements and

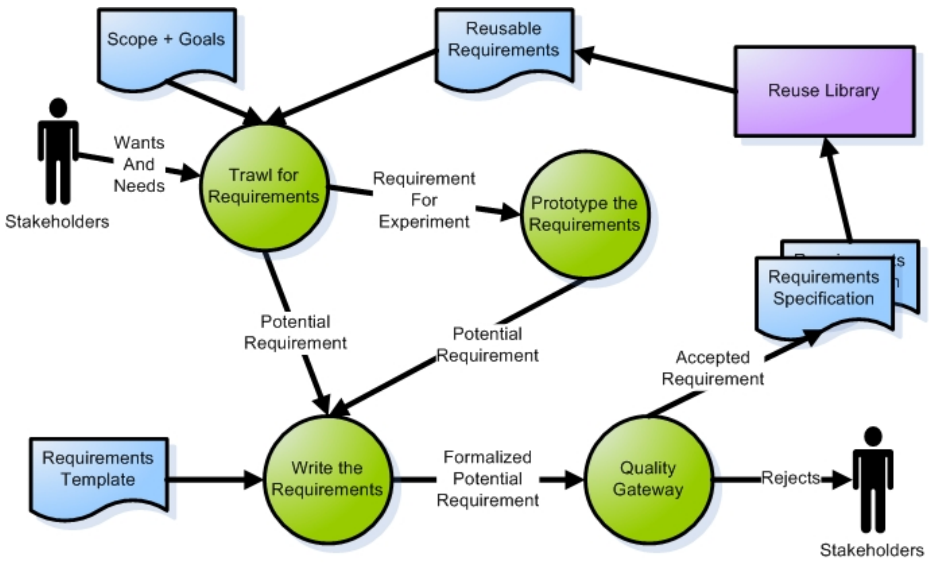


Fig. 2. Overview of the approach from Robertson and Robertson (adapted from [4])

the prototyping of that potential requirements. Each potential requirement will be specified using a template so it will become formalized. The formalized potential requirements will go throw a quality gateway which will reject or accept it. All accepted requirements together are the requirements specification. These steps will be described now in more detail.

Trawling for Requirements The process called *Trawling for Requirements* is about techniques for discovering, eliciting and inventing requirements. After eliciting the functional requirements and specifying the products functionality the non-functional requirements which are the properties the product must have will be discovered and specified using templates. The Robertsons points out that the non-functional requirements describe the qualities of the product - whether it need be fast, or safe, or attractive, and so on. The qualities depend on the functions the product have, the functional requirements. The Robertsons state that NFR can come to light at any time and that they are usually uncovered by the functional requirements. However, the other way around the FR are covered by NFR so they describe a method to ask for each FR what properties or qualities must this piece of functionality have. Their systematical process for finding non-functional requirements is to create a list with all FR and use cases and write to each of them the needed NFR types. To find the corresponding non-functional requirements they suggest to interview the user by using a list of the 8 types of non-functional requirements probing for examples of each type. These types are described below.

Non-Functional Requirements Classification System The Volere template defines 8 types of non-functional requirements. Applying the terminology used in this paper this types can be seen as quality attributes which means that the Robertsons define their own quality model. However, the requirement types are their device for finding non-functional requirements. The proposed practical application is to create a checklist with that types and interview the stakeholders to find potential requirements covered by this types which are as follows:

1. *Look and Feel Requirements* - the spirit of the product's appearance.
2. *Usability Requirements* - the product's ease of use, and any special usability considerations.
3. *Performance Requirements* - how fast, how safe, how many, how accurate the functionality must be.
4. *Operational Requirements* - the operating environment of the product, and what considerations must be made for this environment.
5. *Maintainability and Portability Requirements* - expected changes, and the time allowed to make them.
6. *Security Requirements* - the security and confidentially of the product.
7. *Cultural and Political Requirements* - special requirements concerning cultural and political guidelines which the product must meet.

Prototype the Requirements The Robertsons describe a process called *Prototyping and Scenarios* which use simulation techniques helping to find requirements. Prototypes can be used to drive out the non-functional requirements by giving the stakeholders the opportunity to try out the functions so that non-functional requirements belonging to the categories like usability, look and feel, and security can be determined. At the stage of prototyping a vague description of each elicited non-functional requirements is written.

Writing the Requirement The Volere offers a template which is a container for the whole specification. A further template, called *The Shell*, is a container explicit for requirements which will be used to specify them. The shell contains:

– requirements number: unique identifier
– requirements type: the type from the template
– Event/Use Case numbers: List of events or use case that need this requirement
– Description: One sentence about the intention of the requirement
– and so on.

Having prototypes, consecutively the vague description of the elicited requirements have to be formalized so the next step is to write measurements which quantify the meaning of each requirement. Those measurements and criteria respectively for each requirement are called by the Robertsons the *Fit Criterion* which is described below.

Fit Criterion Robertson and Robertson state that if a requirement can not be measured, then it is not really a requirement. The *Fit Criterion* is an instrument to quantify quality requirements. Again, the 8 types and the prototyping help to find the requirements and at those stages they elicited requirements are documented vaguely. However, not until the Fit Criterion is applied to each elicited requirement they are not measurable and testable. Actually, the Fit Criterion is a formal description of an requirement and is usually derived some time after the requirement description is written down. The idea is for each requirement to have a quality measure that makes it possible to divide all solutions to the requirement into two classes, those for which it can be agreed that they fit the requirement and those for which it can be agreed that they do not fit the requirement. The Fit Criterion is not a test or the design for a test, it is a unambiguous goal that the product has to meet. It can be used as input for a test which will ensure if each products requirement complies with its Fit Criterion. Robertsons points that quantifying the requirement gives a better opportunity to interact with the stakeholders and that both, the requirements engineer and the client, have the same understanding of the requirements. They also suggest that testers should be early involved because they are a good help for specifying the Fit Criterion.

Quality Gateway The *Quality Gateway* is a process that prevents unworthy requirements from entering the specification. It is the entry point of each formalized potential requirement into the requirement specification. If accepted a potential requirement becomes a real requirement. The Quality Gateway applies a number of tests to formalized potential requirements. For example, each requirements will be tested focusing different issues like completeness, consistency, relevancy, and correctness. All requirements which pass the Quality Gateway create together the final requirements specification.

4.4 Approach by Basili, Donzelli and Asgari

The next approach is called *Unified Model of Dependability* (UMD) and is described in [14]. It is focusing on the quality attribute dependability of software systems and is not a complete quality requirements elicitation and specification process in the matter of quality models like the ISO/IEC 9126. Dependability is considered by the authors as a key system or quality property that must be guaranteed regardless of continuous, rapid, and unpredictable technological and context changes. It is defined as trustworthiness of a computing system which allows reliance to be justifiably placed on the services it delivers.

The UMD is capturing and analyzing the impact of negative events (like denial-of-service attacks) on critical system properties such as system safety, availability or security. Different stakeholders will focus on different system attributes like availability, performance, real-time response, and ability to avoid catastrophic failures and resist adverse conditions. UMD helps these stakeholders to build dependability models by providing a structured framework for eliciting and organizing dependability needs. These models identify the measurable and implementable properties that individual systems need to be dependable for their users. For example, performance is a specific dependability attribute of a web application's query service and can be defined as a static or dynamic capability by using metrics like response time or throughput. Based on this the desired behavior can be specified in a performance requirement, for example, the response time should be not greater than 10 seconds. This means a performance failure occurs when the query service responds in more than 10 seconds. Yet, the same failure can also result in a lack of safety (e.g. if the application supports an emergency operator) or in a lack of availability, survivability, security or so on. What we see is that the same failure can be interpreted differently which is a one-to-many relationship between failures and attributes. This implies the need for bottom-up modeling which is covered by this approach.

The UMD uses issues as starting points concerning requirement elicitation. So stakeholders model dependability by defining an issue which should not affect a defined scope of a system until an negative event occurs that cause it. While an event causes an issue, an issue concerns a scope. For example, a denial-of-service event causes a response time greater then 10 seconds which concerns the scope of the query service. Fig. 3 illustrates the structure and the main elements of the UMD method, the concept will be explained subsequently.

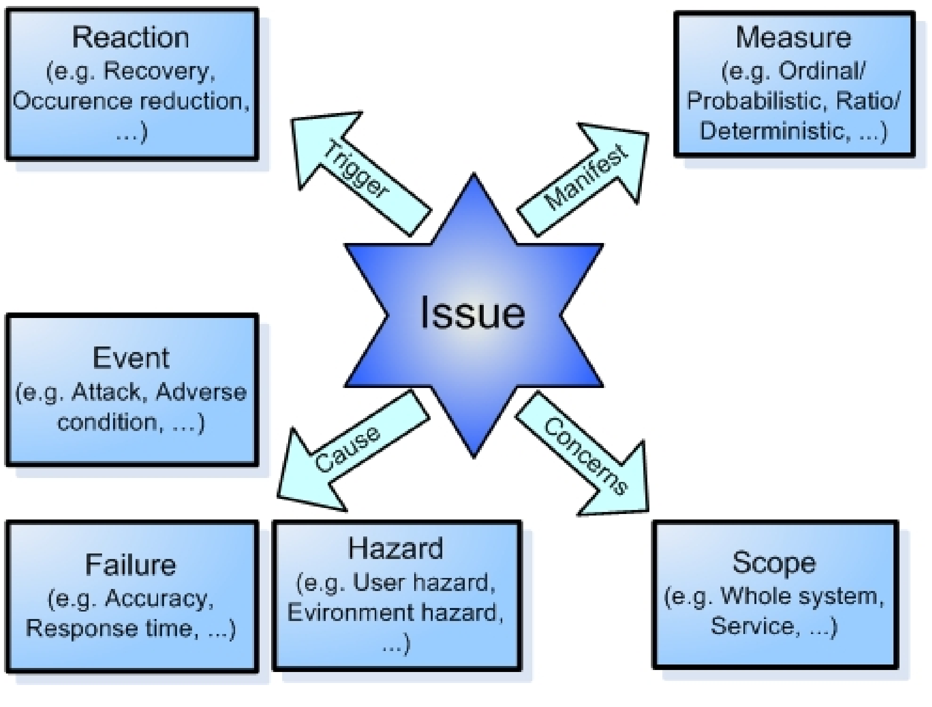

Reaction (e.g. Recovery, Occurence reduction, ...)
Measure (e.g. Ordinal/ Probabilistic, Ratio/ Deterministic, ...)
Trigger
Manifest
Issue
Event (e.g. Attack, Adverse condition, ...)
Cause
Concerns
Failure (e.g. Accuracy, Response time, ...)
Hazard (e.g. User hazard, Evironment hazard, ...)
Scope (e.g. Whole system, Service, ...)

Fig. 3. Structure of UMD (adapted from [14])

Supporting Elicitation The approach of Basili et al. guides stakeholders during the elicitation process and incorporates models, definitions and classifications adopted in the literature. Issues can be categorized into concepts of failures and hazards to help stakeholders identify potential system misbehaviors. The concepts can be further broken down and refined by introducing subclassifications for both failures and hazards. Either ad hoc definitions can be introduced or standards can be applied or adopted for failures and hazards, and also for events and scopes. The method is experience based, so issues, failures, hazards, events, scopes and so on can be reused. The concepts of issue, scope and event are invariant while the mapping of issue to failure is semi-invariant. In addition, the concepts of failure classes, hazards and events are completely customizable.

Measuring Dependability Issues that should not occur have to be specified using an operational definition of dependability. UMD provides measurement models via the invariant concept measure. The stakeholder can choose a measurement style for the definition of dependability. For example, the following measurement classes are suggested: *ordinal and probalistic*, *ratio and probalistic*, or *ratio and deterministic*.

Improvement of dependability UMD is one of few approaches that provides improvement of dependability. It uses the invariant concept of reaction. Stakeholders can suggest reactive or proactive services. Reactive services are triggered by an issue and can warn the user or try to reduce the issue consequences. Proactive services can reduce the probability of an issue's occurrence or can allow a quicker recovery by doing data backups. Classes for reaction which are proposed by UMD are warning services, alternative services, migration services, recovery behavior, and occurrence reduction.

Case Study A feasibly analysis of the UMD concept has been performed within the NASA High Dependability Computing Program. UMD was used to find precisely expressed dependability requirements for a software system designed to aid air traffic controller in detecting and resolving short-term conflicts between aircrafts. Before applying UMD some dependability requirements were elicited by an conventional method so the results can be compared with them. For supporting the process a software tool concerning UMD was used which provides a scope frame and an issue frame. The first step of the case study was the scope definition which means to select the main services of the software and specify it in a scope table. The next step was the model building while each stakeholder filled as many tables as necessary to define his dependability needs. Available characterizations has been used and when necessary extended. In the data analysis phase the tool was used for data visualization based on the completed tables of the data gathering. Analyzing the visualization risk areas could be pointed out. Iterations were performed and ended until stakeholders and analysts felt confident about the results. Finally, data reconciliation has been done, for example when some stakeholders had filled in tables concerning the same service but identified different classes of failures. At the end it has to be said that the applying of UMD brought additional and more precise dependability requirements to light which have not been elicited before.

5 Discussion / Open Issues

All analyzed approaches are comprehensive methods concerning requirements engineering which use structured quality models and quality attributes, and provide methods for elicitation and documentation. The first three approaches concern NFR and not only quality requirements. Furthermore, Robertson and Robertson provide with their *Volere Requirements Process* comprehensive requirements engineering for both types of requirements, FR and NF. The approaches of Doerr et al. and Ebert concentrate more on non-functional requirements then on functional requirements but they suggest to intertwine functional requirements with non-functional requirements. Basili's et al. approach is not designed for common NFR engineering but is focusing in the quality requirement domain of dependability which can be seen a part of quality requirements and non-functional requirements, respectively.

Doerr's et al. method provide the most comprehensive elicitation guidance by using a prioritization questionnaire in a workshop with stakeholders to elicit the primary quality attributes. Based on the elicited quality attributes this approach utilize checklists derived from quality models of these quality attributes which help to elicit NFR in concern with use cases and a high-level architecture. Ebert's approach has no detailed elicitation guidance but separates the requirements in two groups, user-oriented NFR and development-oriented NFR and suggest to elicit them separately with different stakeholders, e.g. users and developers. That separation could be helpful for finding the requirements but the general approach is not as systematically in elicitation and specification as Doerr's approach. Robertson and Robertson's elicitation process base upon the assumption that FR are covered by the NFR. The idea is to create a list with all functional requirements and use cases and ask for each entry what properties or qualities must this piece of functionality have. They also suggest to interview the stakeholder, in particular the user, using their quality model of the 8 non-functional requirement types and probing for examples of each type. Additionally they provide a reuse library of requirements which can be used as a reference while elicitation. Basili et al. utilize issues for the requirements elicitation and incorporates models, definitions and classifications adopted in the literature. Starting with selecting the scopes of the system and functional requirements, respectively, issues are defined for each scope which can be broken down in hazards and failures. The approach is buttom-up and is the only one analyzed in this paper which uses issues. By defining issues for each pre-selected scope and FR respectively the possible failures and resulting hazards will bring additional NFR to light. Issues seem to be a very interesting possibility for quality elicitation because it is a systematically way to find new requirements.

Additionally applying this method in other NFR elicitation approaches could be an advantage.

All approaches provide templates for the documentation of the NFR. The Robertsons provide fix templates which can be obtained from their website. Ebert use domain- and project-dependent templates in his approach and Doerr provides customizable and experienced based templates which will be adapted in every project. Basilis et al. suggest to use templates from the literature. The measurement of NFR is intended in all approaches by providing experience-based reference models concerning metrics. The Robertsons and Ebert constrain the requirement engineer to specify a criteria and fit criterion respectively for a specified non-functional requirement. Doerr uses experienced-based quality models which will be tailored in each project and refined until their sub categories are on metric level. Basili propose different measurement characterizations that can be selected for an certain issue. Yet, no approach gives a comprehensive algorithm how to specify the metric for a certain quality attribute but help by providing reference metric characterizations or refinement of quality attributes in quality models down to metric levels.

The integration of the subjects of functional requirements, non-functional requirements and system architecture is concerned in all methods, especially in Doerr's and Ebert's approaches. Of course, the integration is very important because the three subjects have mutual dependent aspects, actually, they have even impact on each other. Furthermore, Ebert's is the only approach which propose commercial analysis of the elicited non-functional requirements. Yet, it can be discussed if cost aspects should play a more important role in all approaches. An further aspect of Ebert's approach is the separation of user-oriented NFR and development-orient NFR which maybe helpful for finding NFR, comparable to the divide and conquer principle. He also impose the design of roadmaps for NFR which might be an interesting instrument under project managing aspects, in particular time management.

The approach of Basili et al. is limited on the quality attribute dependability but provides extended and detailed examination and coverage for this attribute. By working not only top-down but rather bottom-up the effect of issues to scope can be used to find a more complete set of NFR concerning dependability. It might to considered to introduce the concept of issues in other quality attributes. Further, the only approach that envision the use of CASE-Tools is the UMD. A tool was used in the cases study concerning UMD that provides an interface to input scopes and issues and provides an visualization feature which helps engineers to analyze the requirements in a more convenient way.

Finally, nearly all methods combine concepts like quality models, non-functional requirements, experience, iteration, and tailoring. Open issues in all the approaches are that there is no explicitly implying of assignment from stakeholders and activities to quality requirements. Further, the treatment of cost aspects concerning quality requirements have to be discussed in more detail.

6 Conclusions

There is a wide spectrum of approaches concerning non-functional requirements or quality aspects in industries and literate. Yet there is no approach that exclusively deals with generic quality requirements engineering. However, quality requirements are part of non-functional requirements so the approaches are adaptive for this domain. Concerning non-functional requirements so far no standard approach has been crystallized. However, all approaches use the concepts of quality models and quality attributes to define quality. Actually, they try to give guidance in elicitation by using reference models and reuse libraries. Doerr's and Basili's approaches also allow to use standard quality models like the ISO/IEC 9126. However, the Robertsons as well as Ebert introduce their own proprietary quality models. There is no accepted standard way how to make a non-functional requirements measurable so the approaches force the requirements engineer to refine specifications of the elicited requirements until they are measurable. No approach intertwine explicitly activities and stakeholders with selected non-functional requirements. Cost and benefit analysis are not explicit considered in all approaches.

In summary it can be said that all analyzed approaches provide elicitation and specification of non-functional requirements. Volere is the most comprehensive requirements engineering method which can be used as a stand-alone method. It has to be kept in mind that this method is proprietary and uses his own concepts, for example the quality model or the requirements templates. Ebert's and Deorr's approach can be integrated in other requirement engineering methods and are more adaptable. Actually, Doerr's et al. approach is the most detailed and systematically approach concerning elicitation and specification of non-functional requirements. Basili's approach is very interesting and has a lot of potential due to the use of issues concerning non-functional requirements but it is for now limited to the quality attribute dependability. It should be considered to expand the method to an level that covers all types of quality requirements, and not just dependability requirements.

References

1. Boehm, B., In, H.: Identifying quality-requirement conflicts. IEEE Software **13** (1996) 25 – 35
2. Ebert, C.: Dealing with nonfunctional requirements in large software systems. Ann. Softw. Eng. **3** (1997) 367–395
3. Jacobs, S.: Introducing measurable quality requirements: a case study. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering, IEEE Computer Society Press (1999) 172–179
4. Robertson, S., Robertson, J.: Mastering the Requirements Process. ACM Press, Addison-Wesley (1999)
5. Wagner, S., Deissenboeck, F., Winter, S.: Managing quality requirements using activity-based quality models. In: 5th Workshop on Software Quality (6-WoSQ), ACM Press (2008)

6. Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., Girard, J.: An activity-based quality model for maintainability. In: IEEE International Conference on Software Maintenance, IEEE Computer Society Press (2-5 Oct. 2007) 184–193
7. Wagner, S., Deissenboeck, F., Winter, S.: Erfassung, Strukturierung und Überprüfung von Qualitätsanforderungen durch aktivitätenbasierte Qualitätsmodelle. In: Erhebung, Spezifikation und Analyse nichtfunktionaler Anforderungen in der Systementwicklung. Halbtägiger Workshop in Zusammenhang mit der SE Konferenz 2008. (2008)
8. ISO: ISO/IEC 9126-1, International Standard Information Technology, Software Engineering - Product Quality Part 1: Quality Model (2001)
9. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers (2000)
10. Cysneiros, L., Leite, J.: Driving NFR to use cases and scenarios. In: XV Brazilian Symposium on Software Engineering. (2001)
11. Deissenboeck, F., Wagner, S., Pizka, M.: Kosten-basierte Klassifikation von Qualitätsanforderungen. In: Erhebung, Spezifikation und Analyse nichtfunktionaler Anforderungen in der Systementwicklung. Halbtägiger Workshop in Zusammenhang mit der SE Konferenz 2007. (2007)
12. Paech, B., Kerkow, D.: Non-functional requirements engineering quality is essential. In: 10th International Workshop on Requirments Engineering Foundation for Software Quality. (2004)
13. Doerr, J., Kerkow, D., Koenig, T., Olsson, T., Suzuki, T.: Non-functional requirements in industry - three case studies adopting an experience-based NFR method. In: 13th IEEE International Conference on Requirements Engineering, IEEE Computer Society Press (29 Aug.-2 Sept. 2005) 373–382
14. Basili, V., Donzelli, P., Asgari, S.: A unified model of dependability: Capturing dependability in context. IEEE Software **21**(6) (2004) 19–25

Detection and Fixing of Model Inconsistencies

Frieder Pankratz

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
pankratz@in.tum.de

Abstract. In software engineering processes and development, software models are used for an abstract view on the project. In these models inconsistencies can arise, in the model itself or between two different models. This paper will present three different approaches on how to detect inconsistencies in a model. It will also give a brief overview of the development support tool implementations of these approaches. The third section of this paper is about how to fix an inconsistency, once located, and which problems are related to fixing an inconsistency. In the end the advantages and disadvantages of the three approaches will be discussed.

1 Introduction

The more complex a project is, the more difficult it is to maintain the quality standards for the documentation and implementation. In large software projects, with hundreds of developers working on different but related models representing parts of the same system specification, the system is bound to contain inconsistencies among the models, because of the distributed development and the number of people involved [1]. The same holds true for collaborative projects involving developers with different areas of expertise. Here, not the number of people working on the same document are the origin of the inconsistencies but they might have different conflicting viewpoints on the matter or process deviations.

Models define an abstract view on the system. These models are for example data models, architectural models or models describing the functionality of the program. Different models are used to specify the different parts of the system. In the beginning the models where created by hand, but now almost all models are created using some tool. The tools themselves might even prevent you from constructing inconsistent model elements, but their capabilities to keep the model overall consistent are not enough. If a development technique like "model driven development" [2] or "model based testing" (pages 35ff.) is used, which heavily depends on the model, then even a minor inconsistency can lead to a major failure either during code generation or during runtime and reduce the quality of the software product heavily. It is inefficient and sometimes even impossible to find and keep track of these inconsistencies by hand. Therefore this paper will cover three different approaches about how to detect and fix inconsistencies in a model.

This paper will mainly focus on UML [3] for defining a model, since the UML standard defines different diagram types with their semantics. Each UML diagram type has its own meta-model and UML also defines a meta-meta-model. UML defines diagrams for the static and dynamic structure of a program. For the static structure of a program diagrams like class diagrams, component diagrams or deployment diagrams are used. The dynamic structure of a program is represented in diagrams like the state machine diagram, sequence diagram or use case diagram.

Since UML defines many different diagram types, some information between the diagrams might overlap. These overlapping information is wanted, to create relations between the diagram types, but it can cause inconsistencies if the information from the different diagrams contradict each other. An example for inconsistency between different models is illustrated in figure 1. The class *Streamer* is defined in a class diagram and contains two functions. The associated state chart describes the behavior of the class, that is, in which order the functions of the class are allowed to be called. These models are inconsistent since the class does not contain a function *pause()* or the state machine uses the function *pause()* as a transition which is not defined in the class. An inconsistency can also be within a single model. In this case the model does not follow the UML specification or the forms of best practice, if one wants to use that as consistency criteria. For example figure 2 shows a use case with the actor being within the system. This is generally not allowed and can be considered an inconsistency.

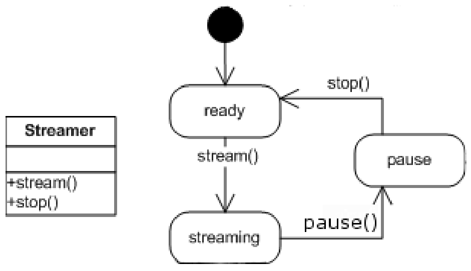


Fig. 1. Class description and associated state chart

To detect such inconsistencies the three approaches, discussed in this paper, will now be presented.

The first approach by Alexander Egyed [4] will extract the information about a model through its meta-model and use specific consistency rules(which know about the different model elements used and their semantics) to check the model for inconsistent elements. In this paper this approach will be referred to as “Egyed”.

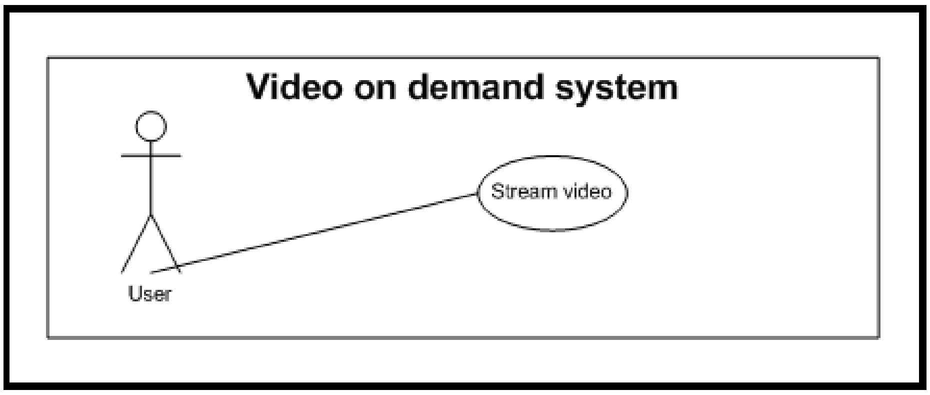


Fig. 2. Use case diagram with an actor within the system

Holger Rasch and Heike Wehrheim [5] developed the second approach, which aims at a formal definition of consistency. A formal language is used as a common semantic domain for the different models. Definitions of consistency are then made on this semantic domain. Referred to as “Rasch/Wehrheim” in the remainder of this paper.

The last approach by Xavier Blanc, Isabelle Mounier, Alix Mougenot and Tom Mens [6] relies on elementary model construction operations instead of the model elements themselves, to be independent of any meta-model. The meta-meta-model of UML will be used as the basis for information gathering. The rules for consistency are defined on the sequence of elementary model construction operations. This approach will be referred to as “BMMM”.

The next section covers the different techniques used for detecting inconsistencies and their development tool implementations. In section 3 ways for fixing inconsistencies and problems like dependencies between inconsistencies will be discussed. The different approaches will be compared in section 4. The last section concludes this paper.

2 Detecting an Inconsistency

The general approach to detect inconsistencies is illustrated in figure 3. To detect an inconsistency, information about the model must be obtained through some way. Rules/definitions for consistency must be evaluated on these information. The evaluation will be performed by a component, in the remainder of this paper called “check engine”. The check engine uses the obtained information and the defined rules or definitions for consistency and verifies whether the information is consistent or not. Depending on the quality of the informations and the rules/definitions for consistency, the check engine is able to name the model elements that caused the inconsistency and provide further information that might be useful for fixing these inconsistencies. There are two basic concepts to detect an inconsistency in a model. The first is to directly use the model as the basis for information and to define the consistency rules/definitions onto the model. The second is to transform the model into an different (formal) representation and to define the consistency rules/definitions onto the (formal) representation.

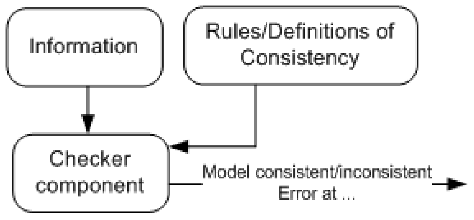

Information
Rules/Definitions of Consistency
Checker component
Model consistent/inconsistent
Error at ...

Fig. 3. General approach for detecting inconsistencies

2.1 Approach by Egyed

This approach extracts the needed information about the model through its meta-model. The check engine uses specific consistency rules, meaning that the rules know about the structure and the semantics of the elements in the model. An instance of each rule is applied to the each model element, if the rule can be applied. It determines wherever the model is consistent or inconsistent. If there are model elements that cause the model to become inconsistent then these model elements are automatically identified through the evaluation process. The rule instance that determined an inconsistency is bound to a model element and hence the inconsistency can be located. A model profiler is used to reduce the reevaluation of the model to the rules that where affected by some change [7]. A model profiler is much like a source code profiler. Instead of observing the source code during execution, a model profiler observes the evaluation of a model during consistency checking (i.e., it knows what fields of what model element are accessed when and how often) [4]. In the case study in [4] no special language was used to define the consistency rules, they where hard coded into the check engine instead. But the design of the tool would allow to replace the hard coded rules and the check engine with some form of rule language and a modified component.

The same example as in [4] will be used to further explain this approach. In Figure 4 the four UML diagrams describing the example model are displayed. The model represents a simplified video on demand system. The class diagram represents the static structure of the system. The class *Display* represents the component used for displaying the video content and receive user input. The class *Streamer* is used for downloading and decoding the video content. The state machine diagrams describe the behavior of the two classes when they receive input. The sequence diagram contains the interactions between the instances of the classes and the user. First the user has to select a specific content, symbolized by the message *1: select*, then the user starts the stream by invoking the message *2: stream* on the object *d* of class *Display*. This in return invokes the message *3: stream* on object *s* of class *Streamer*. The sequence diagram also shows that when the user invokes the message *4: stop* on object *d*, to stop the playback, the message *5: wait* is sent to object *s*.

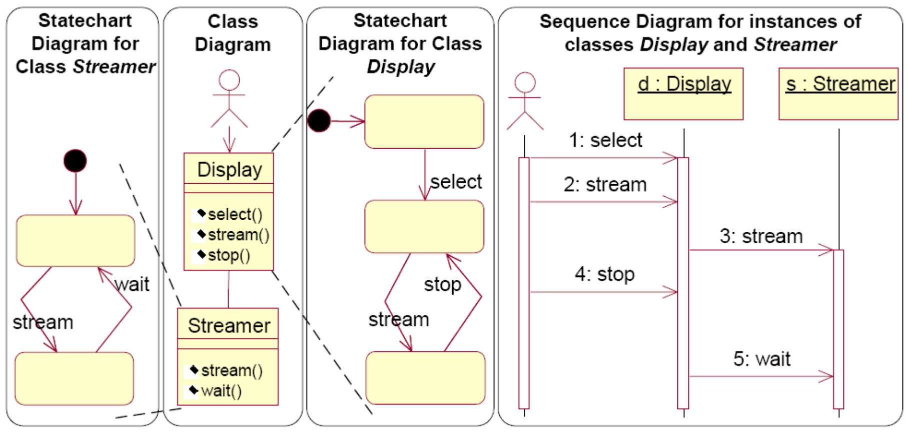


Fig. 4. UML model illustration of an video on demand

Consistency Rules The consistency rules describe conditions that all models must satisfy for them to be considered valid. Rules covering class, sequence and state machine diagrams where defined. These rules where for example:

Rule 1: message name must match class method

Rule 1 states that the name of a message in a sequence diagram must match a method name in the receiver's class [4]. If this rule is evaluated on message*1: select* in the sequence diagram then the receiver(see arrowhead of message) of this message is identified first. The receiver of the message would be object *d*. The base class of object *d* is the class *Display*, containing the set of methods *select*, *stream*, *stop*. Since the message *select* is part of the set of methods of class *Display* the condition will return true. This is a rule to check the syntactic consistency of the model since only static information is needed.

Rule 2: message sequence must match behavior

Rule 2 states that the sequence of incoming message in an object of a sequence diagram must match the allowed behavior of the state machine diagram of the object's class [4]. For example, object *d* receives the messages *select*, *stream* and *stop* - in this order. The state machine diagram of class *Display*(the base of object *d*) allows this behavior because it is a valid sequence. This rule performs a semantic check on the model since the state machine must be simulated to know if the message sequence is a valid run of the state machine.

Development tool support This approach is the only one in this paper that is able to provide choices for fixing an inconsistency. The tool implementation of Egyed was evaluated in a case study using 48 small-to-large-scale UML models covering a total of 250,000 model elements and over 400,000 separate rule evaluations using 34 consistency rules [4]. The tool implementation was integrated into the design tool IBM Rational Rose^{TM}.

An average of 5.4(between 2.5 and 8 in average) choices per inconsistency where made by the tool during the case study. During the case study the scalability of the tool implementation was determined to be linearly complex(both memory and computation cost) and thus quite scalabe [4]. The case study was performed on a 1.7Ghz Intel Centrino processor. The costs for consistency checking where 9 ms per change in average, with a worst case of 2 seconds [4]

2.2 Approach by Rasch/Wehrheim

The aim of this approach is to give a formal definition of consistency. For that the models are translated into a common semantic domain. A formal language is used to define the common semantic domain. The definitions of consistency are made on this semantic domain. Here the information about the models is obtained by the translation. This is done by giving the formal semantic for class definitions and state machines, as these are the only two diagram types covered by this approach. Also Object-Z [8] is used instead of UML [3] to specify the class definitions, because UML does not prescribe a fixed syntax for attributes and methods of classes. The common semantic domain is the failures-divergences model of the process algebra CSP [9]. CSP simulates processes, hence the classes from the class diagram with their state machines are translated into processes. To check the model for consistency the class definitions and the state machines are first translated into CSP. This gives two different views on a class. One view describing the attributes of the class and the possible effects of method execution, derived from the class definition. The second view is derived from the state machine, describing in which order the methods are allowed to be executed. Every element that should be included in the consistency check must be translated into CSP. The formalism to translate state machines into CSP are only known for restricted classes of state machines. The check engine in this case would be the commercial product FDR (Failures/Divergence Refinement) developed by Formal Systems Europe. Two formal definitions of consistency would be:

Definition 1:

A specification consisting of an Object-Z class and an associated state machine has the property of satisfiability iff the corresponding process in the semantic model has at least one non-terminating run [5].

Definition 2:

A specification consisting of an Object-Z class and an associated state machine has the property of basic consistency iff the corresponding process in the semantic model is deadlock free [5].

To explain further how inconsistencies are found using this mechanism, understanding CSP is required which is out of the scope of this paper. For further information please read [5].

2.3 Approach by BMMM

Information about a model is extracted in a different way by this approach than by the other two. Every model can be represented by a sequence of elementary construction operations. So instead of extracting the information out of the model using it's meta-model, the sequence of elementary construction operations is obtained ans stored in its own model. This way consistency can be checked between different kinds of models, regardless of their meta-model. The advantage of this approach is that not only different kind of models can be processed in a uniform way, but also structural and methodological consistency rules can be expressed uniformly as logical constraints on sequences of elementary construction operations. A structural consistency rule would be like rules of "Egyed". They define relationships between model elements. A methodological consistency rule on the other hand are constraints over the construction process itself [6].

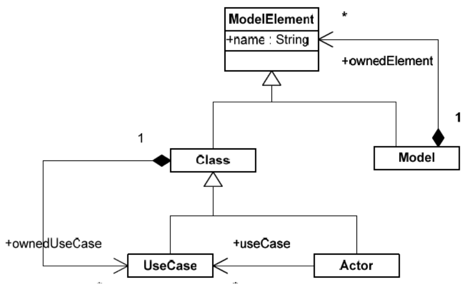
Inspired by the MOF reflective API they defined four elementary operations to construct their model:

– *create(me,mc)* Create a new model element *me* of meta-model class *mc*
– *delete(me)* Delete the model element *me*
– *setProperty(me,p,Values)* Set the property of model element *me*, property *p* with the values in *Values*
– *setReference(me,r,References)* Set the reference *r* of model element *me* to the model elements defined in *References*

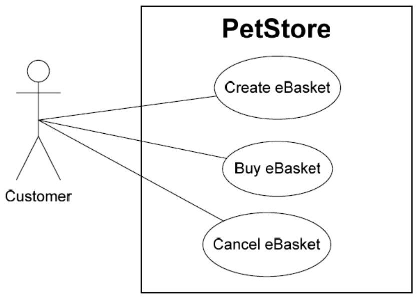
The same example as in [6] to illustrate the mechanisms of this approach will be used. In Figure 5 (a) a simplified part of the UML 2.1 meta-model for use cases is illustrated. This meta model will be referred to as UCMM for Use Case Meta Model in the rest of this paper. Figure 5 (b) contains a use case based on the UCMM. It is composed of an actor (named "Customer") that is associated with three use cases (named "Create eBasket", "Buy eBasket" and "Cancel eBasket") belonging to a class (named "PetStore") representing the system.

Using the previously defined construction operations, the use case model in figure 5 (b) can be constructed using the following construction sequence σ:

The meta-model classes "Class", "UseCase" and "Actor" can be found in the UCMM class diagram as the class names. The properties "name", "ownedUseCase", "useCase" are derived from either class attributes or from relations between the classes.



(a) The UCMM class diagram



(b) A use case model instance of UCMM

Fig. 5. UML model illustration of the UCMM example

1. create(c1,Class)
2. setProperty(c1,name, "PetStore")
3. create(uc1,UseCase)
4. setProperty(uc1,name, "Buy eBasket"))
5. create(uc2,UseCase)
6. setProperty(uc2,name,"Create eBasket")
7. create(uc3,UseCase)
8. setProperty(uc3,name,"Cancel eBasket")
9. setReference(c1,ownedUseCase,uc1,uc2,uc3)
10. create(a1,Actor)
11. setProperty(a1,name, "Customer")
12. setReference(a1, useCase, uc1,uc2,uc3)

Fig. 6. Construction sequence σ of use case diagram in figure 5 (b)

To define structural consistency rules three predicates can be defined: *lastCreate*, *lastSetProperty*, *lastSetReference*. The *last* indicates that this operation performed in the current sequence will not be followed by an operation negating this operations. This would be the case for example if a *create(me, mc)* operation is followed by a *delete(me)* operations somewhere in the sequence. These *last*-operations are useful since the structural consistency rules concern themselves only with the model obtained after the sequence of construction operations is performed, hence operations that later get negated are not of interest. A structural consistency rule for example would be:

Rule 1: An actor should not own a use case

This structural consistency rule is inspired by the UML 2.1 specification that states that an actor should not own a use case even if the meta model permits it. For our sequence σ in figure 6 it would mean that all *create(me, Actor)* operations, to create actors, must not be followed by a *setReference(me, ownedUseCase, References)* operations. The *last*-operations can be used in the following logic formula to keep things more simple and efficient:

```
ActorsDoNotOwnUseCase($\sigma$)=true iff
  $\forall$ a $\in \sigma$,
    if a = lastCreate(me,Actor) then
      $\nexists$ o $\in \sigma$
        o=lastSetRefernce(me,ownedUseCase,R) and R $\neq \emptyset$
```

In our example in figure 6 at operation 10 a new actor is created. After that no *setReference(a1,ownedUseCase,R)* occurred, hence our example is consistent.

As methodological inconsistency rules constrain the construction process, the order between operations should be taken into account during the evaluation of the rules [6]. Therefor we define:

Let $\sigma = a_1; ...; a_{m-1}; a_m; a_{m+1}; ...; a_n$ be a construction sequence.
We denote by $a_i <_\sigma a_j$ the occurrence of operation a_i before a_j in sequence σ.

Rule 2: A class must exist before a state chart, describing the class, can be created

This methodological consistency rule is based on figure 7. This class diagram introduces a relationship between classes and state charts, a state chart *describes* a class. The intention of this rule is to make sure that first a class is designed using a class diagram and then further specified using a state chart, and not the other way around. Using the construction operations, this can be expressed through: if there is a *setReference(sc, describes, c)* operation, then there must be no *create(sc,StateChart* operation before the *create(c,Class)* operation. As a formula it would be:

```
ClassBeforeStateChart($\sigma$) = true iff
  $\forall a \in \sigma$, if a = setReference(sc, describes, cl) then
    $\exists c \in \sigma$, create(cl, Class)
    and $\exists b \in \sigma$, create(sc, StateChart)
```

and $c <_\sigma b <_\sigma a$

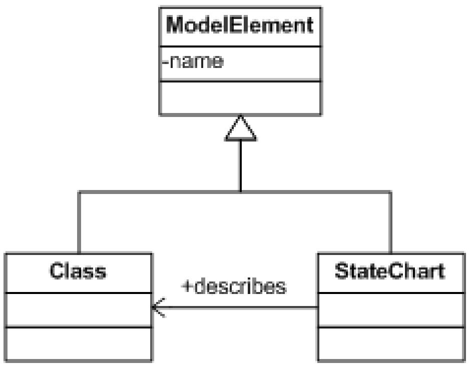


Fig. 7. Class diagram describing the relationship between class diagrams and state chart diagrams

In figure 8 we see the simplified construction sequence of the class and state chart diagrams of figure 4.

1. create(c1,Class)
2. setProperty(c1, name, “Display”)
3. create(sc1, StateChart)
4. setProperty(sc1,name,”StateChart for Display”)
5. setReference(sc1, describes, c1)
6. create(sc2, StateChart)
7. setProperty(sc2, name, “StateChart for Streamer”)
8. create(c2,Class)
9. setProperty(c2, name, “Streamer”))
10. setReference(sc2, describes, c2)

Fig. 8. Construction sequence σ for state chart and class diagram for example 4

The first part with the operations $\{1 - 5\}$ in figure 8 is consistent as the class $\{1\}$ is constructed before the state chart $\{3\}$. The second part with the operations $\{6 - 10\}$ is inconsistent, as the state chart is created before the class.

For the evaluation of this approach the checker engine was developed in Prolog. The sequence of construction operations was stored as Prolog facts, and the rules where translated into Prolog queries by hand. The Prolog queries where implemented as the negation of the consistency rules.

Development Tool Support A prototype was build using Prolog as the programming language for the checker engine and integrated into the modeling environments Eclipse EMF and Rational Software Architect. The prototype has been validated on a real large-scale UML model. 58 constrains where defined for the test. The model contained about 1.3 million construction operations. 45 seconds where needed to load the model into the memory and 3 minutes where needed for checking the 58 consistency rules using an Intel Pentium D CPU 3.00GHz with 3 GM ram memory.

3 Fixing an Inconsistency

An inconsistency can not only arise from human error but also from desired design changes or from fixing an inconsistency. Since an inconsistency is not an independent event, they either share erroneous model elements or the erroneous model elements are located in close proximity in the model [4]. Because of these dependencies among the inconsistencies, fixing an inconsistency can have desired and undesired side effects.

The approaches by Rasch/Wehrheim and BMMM only offered ways to detect inconsistencies, but Egyed also offers a way to analyse these dependencies and give reasonable suggestions and hints for fixing inconsistencies.

Egyed detects inconsistencies by creating an instance of each rule for every model element(only if the rule can be applied to the model element), like a class or a message. If these rules detect an inconsistency then the corresponding model element introducing the inconsistency is identified through the rule instance. By using the model profiler and further investigating the scope of the rule that detected the inconsistency, the cause of the inconsistency can be further isolated. The scope of a rule are the model elements and attributes that must be accessed to evaluate the rule. By further investigating the scope of an inconsistent rule all dependencies to other rule instances can be obtained. Basically, fixing an inconsistency is the same as a design change. A design change may cause one or more of the following situations. A new rule instance might be created and the rule evaluates itself to be either consistent or inconsistent. A reevaluation might occur on some rules. These rules may stay in their current state (consistent/inconsistent) or may change. Also a design change can cause an rule instance to become obsolete and be removed.

An example will be used to further explain the process for fixing an inconsistency. As a basis, the model that was used to explain Egyed(see figure 4) will be used. Assuming the designer of the video on demand system decided to rename the *stream* method of the class *Display* to *play*, to reduce confusion since the same method names are used in the two classes. He did not only rename the method, he also renamed the state transition in the *Display* state machine diagram and the message *3: stream* in the sequence diagram. Unfortunately the designer renamed the wrong message in the sequence diagram, he should have renamed the message *2: stream* and because of that multiple inconsistencies are introduced. The rules will now be reevaluated and the following rule instances have become inconsistent. The first column in table 1 introduces a name for the rule instances, the second column gives information about with rule was instantiated and the last column describes to which model element the instance was assigned.

Table 1. Inconsistent rule instances

Identifier	Rule	Rule location
R11	Rule 1	evaluated on link message *2: stream()*
R12	Rule 1	evaluated on link message *3: stream()*
R21	Rule 2	evaluated on object *s:Streamer*
R22	Rule 2	evaluated on object *d:Display*

R11 became inconsistent because the class *Display* no longer contains a method with the name *stream*. R12 became inconsistent because the message *play* is not included in the methods of the class *Streamer*. R21 and R22 became inconsistent for the same reasons. By a small mistake four inconsistencies where introduced. By further investigating the the scope of the rule R12 using the model profiler we can identify all elements that could cause this inconsistency and derive from those information ways to fix it. Possible fixes could be:

1. renaming message *play* to *stream*
2. changing the receiver of the message *play* to object *d*
3. adding a new method *play* to the class *Streamer*
4. change the ownership of object *s* to class *Display*
5. rename method *play* to *stream*
6. delete the message *3: stream*

There are more possible solutions to fix this inconsistency and not all of the presented fixes really solve the problem. The tool can't automatically decide which fix should be applied, only the designer has enough knowledge to decide which fix should be used. So changing the class of object s from *Streamer* to *Display* would solve the inconsistency of rule instance R11 but application would not be able to function any longer. The designer must be the one to decide how to fix an inconsistency since application design is more then just the consistency of the model. Fixing the inconsistency of rule R12 by renaming the message *3: play* back to *3: stream*, fixed also the rule instance R22 as a side effect. In [4] inconsistencies have dependencies if they share at least one possible fix. Egyed also determines to some extent which side effects could occur if a specific fix would be used. Using this information the the tool is able to categorize the offered choices for fixes into four categories.

1. **Unused Choices do not have Side Effects**
   These fixes for a inconsistency have no side effects because the fix affects only one model element.

2. **I-Only Choices have Positive Side Effects Only**
   A fix marked as I-Only affects multiple rule instances that are currently inconsistent but no rule instances that are currently consistent.
3. **C-Only Choices have Negative Side Effects Only**
   These fixes may affect consistent rule instances, but this is not guaranteed since false positives are possible.
4. **C&I Choices have Positive & Negative Side Effects**
   A choice marked C&I affects both inconsistent and consistent rules.

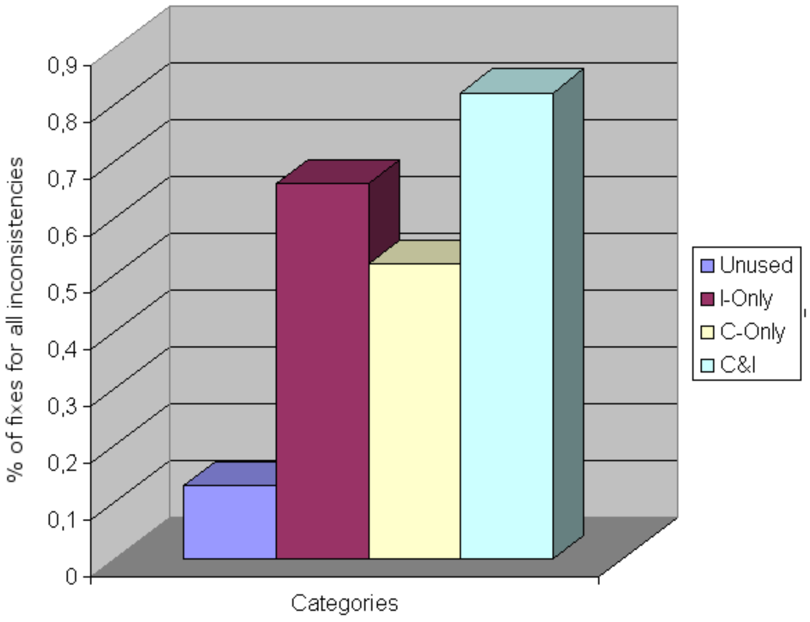


Fig. 9. Histogram of the fixing choices of the four categories

Every inconsistency has more then just one possible fix, which is why the percentage numbers in figure 9 don't add up to 100%. For further information please read [4].

4 Advantages and Disadvantages

Since the purpose of each approach is different, comparing the three approaches with each other might create a wrong impression. Egyed tried to develop a system capable of providing real time feedback, mostly for syntactic inconsistencies. The approach by Rasch and Wehrheim tried to give a formal definition of consistency, using a model checker from a related problem domain. The goal of BMMM's approach was to include the construction process of a model into the consistency checks, enabling methodical consistency checks. In this chapter the advantages and disadvantages of each approach will be discussed.

All three approaches have in common that all needed information about the model is extracted through the model. No special annotations in the model are necessary to enable the translation or to extract the information. Also all three approaches where able to define syntactic consistency, but not all provide support for semantic inconsistencies. To provide support for semantic inconsistencies, for example between a sequence diagram and a state chart, the state machine has to be simulated, to make sure that the sequence of messages in the sequence diagram is allowed. Egyed's and Rasch/Wehrheim's approach allow such consistency checks, without further development of the approach.

Even though the rules are hard coded into the checker engine in Egyed's approach, the design of the application allows an easy replacement of that component without any restraints. Since this approach was developed with evaluation speed in mind, the evaluation is fast enough (9ms - 2sec) to provide real time feedback to the designer, as soon as he begins to change the model. Another advantage is that it is able to provide choices for fixing the found inconsistencies and categorize these choices so that the designer knows of any possible side effects. Even scalability is linearly complex with the size of the model. Therefore a tool implementation of this approach could be used in practice and not just as an academic test.

In the approach by Rasch and Wehrheim all consistency definitions are transformed into semantic definitions since the FDR is like a model checker. Which is why this approach is the best for semantic definitions of consistency. But since all model elements must be translated into the semantic domain, a lot of work is needed to extend this approach to other models.

The approach by BMMM was the only one able to define methodical consistency rules. But to do this, the history of the model construction is needed. Another strong point of BMMM is that all consistency rules can be applied uniformly. There is no distinction between the different model types and their meta-models since they use their own model to store the information. Since their model is inspired by the MOF, which defines the meta-meta-model of UML, it is easy to include the UML diagram types into this approach. For other models, a transformation into their model must be possible. As a drawback the consistency rules are more complex to define than the ones defined by Egyed. This is because Egyed uses the meta-model as the basis for his information gathering, hence the semantic information of the classes in the meta-model can be used to define the consistency rules. As in BMMM's approach, the meaning of the sequences must be reconstructed to express the constraints.

5 Conclusion

Inconsistencies can arise within a model or between different models from different situations. This paper demonstrated three completely different approaches for detecting inconsistencies. Egyed used the UML meta-model and a model profiler to gather information and specific consistency rules to evaluate the model. Rasch/Wehrheim tried to give a formal definition of consistency by transforming the model elements into the process algebra CSP and specifying consistency definitions on those CSP elements. The last method by BMMM used its own model to gather all instruction needed to construct the model in question and defined consistency rules on those sequences of instructions. All three approaches where able to detect inconsistencies and name the model elements causing them. But detecting inconsistencies alone is not enough. Inconsistencies have dependencies to other inconsistent or consistent model elements and fixing an inconsistency can cause more problems than it might fix. So if a tool for model consistency is developed, it must inform the the designer not only about the inconsistencies in the model but also about the dependencies of them. If not, then in large projects, in which a tool support like this is needed the most, the designer is most likely overwhelmed by the number of found inconsistencies. A proper post processing of the information is needed to determine the origin of each inconsistency, group inconsistencies according to their origins and maybe even provide possible fixes of the inconsistencies. In this paper only the approach by Egyed was able to provide such information. There are many other approaches on how to detect inconsistencies, for example [10, 11], but few of them go much further then on the detection. I think it's equally important to refine the gathered information about inconsistencies and think of ways to present this information to the designer.

References

1. Briand, L.C., Labiche, Y., O'Sullivan, L.: Impact analysis and change management of UML models. In: ICSM '03: Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (2003) 256
2. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5) (2003) 19–25
3. OMG: Unified modeling language specification v2.0. Technical Report 05-07-04, Object Management Group (2005)
4. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 292–301
5. Holger Rasch, H.W.: Checking consistency in UML diagrams: Classes and state machines. In: Formal Methods for Open Object-Based Distributed Systems, Berlin, Germany, Springer Berlin / Heidelberg (2003) 229–243
6. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE '08: Proceedings of the 13th international conference on Software engineering, New York, NY, USA, ACM (2008) 511–520
7. Egyed, A.: Instant consistency checking for the UML. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 381–390
8. Smith, G.: The Object-Z specification language. Kluwer Academic Publishers, Norwell, MA, USA (2000)

9. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
10. Simmonds, J., Bastarrica, M.C.: A tool for automatic UML model consistency checking. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM (2005) 431–432
11. Kotb, Y., Katayama, T.: Consistency checking of UML model diagrams using the xml semantics approach. In: WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web, New York, NY, USA, ACM (2005) 982–983

Model-Based Testing

Philip Preissing

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
preissip@in.tum.de

Abstract. This work gives an overview of model-based testing, which aims at automatically generating test cases for a software system from a formal model of its intended behavior. It first describes the theoretical foundations, including different modeling languages and different test generation techniques. After that, the practical use of model-based testing is examined. For this purpose, several case studies are described. Here, a focus lies not only on the different application domains, but more on empirical results that help to evaluate the technique and compare it to other quality assurance techniques. The paper concludes with an overview of the academic and commercial tool support for model-based testing. A practical example is conducted using one commercial tool.

1 Introduction

Models play a key role in software engineering. They are, among other things, extensively used to describe the structure and behavior of a software system on a more abstract level than code, thus providing a common understanding between all stakeholders. With the rise of Model Driven Development (MDD), models are now being put in even wider use by employing them in the whole development process, e.g. by generating artifacts out of them or automatically checking consistency between model and code. Considering this trend, it seems straight-forward to use (probably already existing) models for (semi-)automatic test case generation.

A test case compares intended and actual behavior of a software system thereby proving that the system does not operate as expected or increasing the confidence that it does. Test cases are usually designed manually. This process is often undocumented, not systematic, non-repeatable and depends on the knowledge of few test experts. Model-based testing, in contrast, aims at generating test cases from an explicit behavioral model, e.g. a state machine. In a nutshell, the idea is to automatically collect so called *model traces* (an input to the model and its corresponding output). Since the model describes the intended behavior of the system, these traces can then be used as test cases for the real *system under test* (SUT).

Today, testing is probably the most widely applied quality assurance technique. Software firms put great effort in test generation and testing. While testing is often automated, the test case generation has still to be made by hand. Model-based testing promises to heavily reduce this effort without loss in the quality of test cases. In fact the quality could even increase: when the model is available, many more test cases can be generated automatically than could reasonably be created manually. Another point is that the complexity of software systems increases steadily. This influences the complexity of test cases making manual generation of tests even more time-consuming.

Yet model-based testing is no push-button technology and there is still much research effort necessary, not only to improve the technique itself but more importantly to examine whether it actually pays off in practice. Some empirical findings regarding this matter are presented later in this paper.

The next section of this paper explains model-based testing in detail. It describes the testing procedure, which types of models are useful, and how test cases are generated out of them. Section 3 presents empirical findings about the practical usefulness of model-based testing. Section 4 presents the current tool support for model-based testing and Section 5 concludes with a summary, open issues and an outlook.

2 Model-Based Testing

This section explains model-based testing in greater detail. It first discusses the general testing procedure and which models are applicable. Approaches how to select interesting tests out of the probably very large set of possible test cases are also presented. The section concludes with a small example that shows how the different parts work together.

2.1 Testing Procedure

Figure 1 depicts the general approach of model-based testing. Note that some argue that every testing is *model-based* because a test engineer that manually derives tests has an implicit model of the system in his mind. However, in this paper, model-based testing does always refer to an explicit model of the SUT's intended behavior.

In a first step, an abstract behavioral model, for example a finite state machine, is created. There are several different approaches for this: It can either be extracted automatically from the source code, may be a reused, already existing model from the software design phase, or can be created separately based on the requirements. These choices and their implications are covered in the next subsection.

The model is then used to create the mentioned *model traces* which are essentially a series of actions together with input and the final output of the

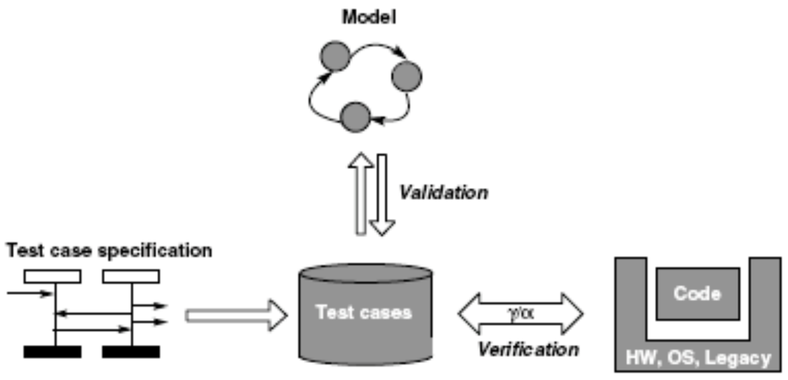


Fig. 1. Model-based testing (from [1])

model. Those traces can be validated to make sure they meet the actual user requirements.

Usually, the creation of test cases happens with respect to so called *test case specifications*. Those are specifications that in some way influence the test case generation, i.e. act as a selection criterion on the set of all possible traces. For example, they can limit the total number of tests by restricting the generation to the most critical parts of a system. This is for example useful if an embedded system is considered that can only handle few test cases in a reasonable timespan.

Before the traces can now be applied to the SUT, it has to be considered that the model is on a higher abstraction level than the source code. Thus, the model traces usually have to be concretized before they can communicate with the real system. For the same reason the SUT's output has to be abstracted before comparing it to the model's output. This process of concretization (γ) and abstraction (α) is performed by a *test driver* or *adapter* which has usually to be customised for each individual software system.

The testing procedure then consists simply in comparing the SUT's abstracted output with the model output: if it matches, the test is passed thereby increasing confidence that the code corresponds to the model, otherwise it fails thereby proving its non-conformance. This is then a form of verification in the sense that it is assured that the actual code conforms to a more or less formal specification encoded in the model.

The main benefits of model-based testing compared with manual test design include ([2]):

- **Cost Savings**: Costs are saved, when the modeling time is smaller than the time necessary for manual test design.
- **Systematic Testing**: Tests are based on test specifications that control the number of tests as well as the desired coverage. Test case generation is repeatable.

– **Modeling benefits**: Through the mere activitity of modeling, requirement/specification errors and ambiguities are detected in an early stage of development where they are still cheap to fix. A model also makes knowlegde explicit and conserves it for future developers and testers.
– **Quick response to evolving requirements**: Changing requirements require simply an adaption of the model. Then the test cases can be regenerated and re-run without any further changing activity, whereas in manual test design a probably high number of existing test cases would have to be adapted by hand.
– **Automated Traceability**: The model provides an automated traceability between requirements and tests. Failures that were found in the code can be traced to model elements and then to requirements.
  This can for example be supported by annotating requirement numbers to model elements.

Note that there is not *the one* model-based testing procedure. Possible variations and their implications in the different steps are discussed in the next subsections. A taxonomy of model-based testing, which aims at providing an overview of the variations, is presented in [3].

2.2 Models

The popular and widely accepted definition of Stachowiak [4] identifies three important properties of models:

1. Mapping: Models are mappings from a real world to a model world.
2. Pragmatism: Models serve a specific purpose.
3. Reduction: Models are simplifications. They are more abstract and do not represent every single aspect of the real world.

Figure 2 illustrates Stachowiaks definition. A model results from some mapping from an original to a model world while some (*waived*) attributes of the original are abstracted away and other (*abundant*) attributes are added. For example, a specification language introduces syntactical elements, e.g. the shape of an arrow used, that have no meaning for the software to be developed. There is no need that the original exists physically; it may be planned or totally fictious.

Models are fundamental for software-engineering: process models, process maturity models, design models and interaction models are part of every software engineer's daily work. Some argue that all programs are itself models of the real world and thus programming is an activity of modeling, too.

When working with models, it is crucial to maintain consistency between all different model artifacts as well as between model and code. This holds especially true when employing models in the test process, where an inconsistent model would probably lead to inconsistent test cases and thereby to tests that fail although the behavior is actually correct. Pankratz (pages 19ff.) describes different approaches how to detect inconsistencies and how to fix them.

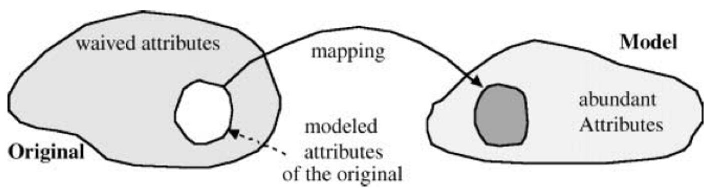


Fig. 2. Mapping between original and model according to Stachowiak (from [5])

Abstraction

The main reasons why models are so widely used in software engineering today, is that they are easier to create, understand, communicate and maintain than their counterparts in the real world. This is mostly due to abstraction, which is achieved by omission and encapsulation of details.

Omission of details means that aspects that seem too detailed or even unnecessary for the model's purpose are abstracted away. Some abstracted information can be reinjected by the driver component whereas other information is lost completely. For example, in a smart card case study [6], cryptographic algorithms were considered too complex and were excluded from the model. The test driver reinjected that functionality before the generated test cases were applied to the SUT.

When some information is encapsulated, models can be made a lot easier to understand by simply referencing this information instead of including it in each individual model. This happens for example in programming languages where stack manipulations are encapsulated in function calls. Another benefit is that this concept of encapsulation also restricts the programmer's possibilities, e.g. the way function calls work cannot be altered. This makes the model more suitable to automatic analysis.

Prenninger and Pretschner [7] differentiate the following different kinds of abstraction used in model-based testing.

- **Functional Abstraction** means to concentrate on the main functionality that has to be verified. The model does not contain the complete intended behavior of the whole system but only significant aspects. For example, exception handling could be abstracted away.

  Funtional abstraction can also be used to divide the SUT's functionality in different parts which can be modeled independently.
- **Data Abstraction** aims at reducing the complexity by using logical or abstract data types instead of concrete ones. This can happen with and without information loss. An example for data abstraction without information loss is encoding binary numbers as integers. Another example is to use equivalence classes of values instead of concrete ones, which does clearly involve information loss. To be able to compare the model output with the SUT's

output, it is necessary to define a mapping between concrete datatypes and abstract ones.

– **Communication Abstraction** is typically necessary when dealing with a distributed system. In the case of an distributed method invocation, a complex communication protocol including handshakes in the SUT can abstracted by a simple atomic operation in the model. During test case generation those simple operations can be substituted with the real interaction procedure.
  But communication abstraction is not limited to distributed systems, it can also be very helpful when testing hardware components that interact using a complex sequence of signals.
  Communication abstraction is often combined with functional and data abstraction.
– **Temporal Abstraction** means that only the ordering of events, not their precise timing is relevant for the model.
  There are basically two forms of temporal abstraction. The first is that model and SUT have a different granularity of discrete time, e.g. a clock cycle in the model corresponds to many clock cycles in the SUT. The other form is that physical time is abstracted, e.g. a timer that lasts for X ms is reduced to two events: one for starting the timer and one indicating it's expiration.
  When also the ordering of events is irrelevant, this is considered communication abstraction.

But abstraction comes with a cost in the context of model-based testing: functionality that was abstracted away cannot be tested. Therefore, the model designer has to trade off between abstraction and precision, so that all aspects worth testing are included. It can thus be infered that there is an inherent complexity in systems that cannot be abstracted away.

A typical example of this is that input parameters in the model are always considered to be in certain bounds. This makes it impossible to test for malformed inputs. When functional abstraction is used to break the system off into different parts, it is hard to detect failures due to interaction between them. Finally, overdosed temporal abstraction can lead to not detecting timing-related failures in real-time components.

Model Redundancy

After reading the preceeding sections, one question might arise naturally: If I already put effort in creating a behavioral model for specification and code generation, is it possible to use the same model for test case generation, since this would save the effort necessary to create another model?

In fact there are more approaches than these two. Figure 3 shows four possible scenarios of model-based testing, ranging from using a common model for both code generation and test case generation to totally separate models.

The first (upper left) refers to using the same model for code generation and test case generation. The second (upper right) describes the automatic extraction of a model from code. The third (lower left) means manually creating a model

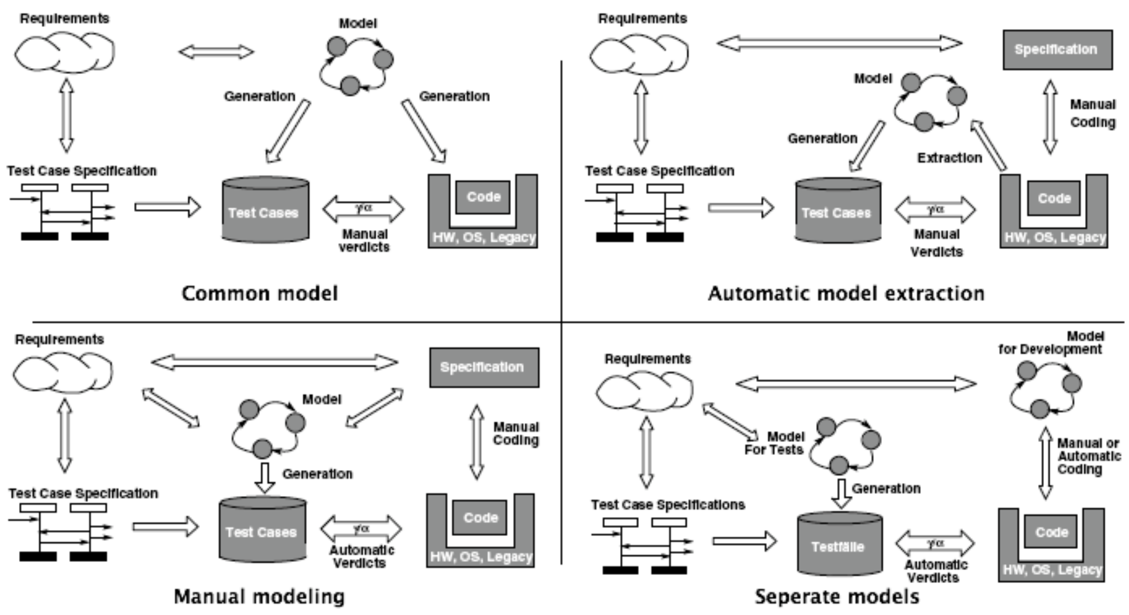


Fig. 3. Scenarios of model-based testing (assembled based on [1])

from requirements and specification. The second and third approach both refer to creating a test model *after* the system was build. The fourth (lower right) approach is concerned with creating two distinct models, one for code generation and one for test generation.

When using the same model for code generation and test generation, there is no redundancy between the intended and actual behavior as both are encoded in the same model. As a result, the code is somehow tested against itself and would conform to all tests if two constraints hold: the code generation worked fine and environment assumptions made in the model are not violated.
These two constraints are the only things that can be tested when using a common model. Nevertheless, this can be also helpful because it explicitly verifies the assumptions made in the model and would find inadequate ones. One tool that aims at this scenario is *Embedded Tester*[1]. It is capable of automatically generating test cases for Matlab/Simulink models. It checks for consistency between all relevant development stages and their models (Model in the loop (MIL), Software in the loop (SIL) and Processor in the loop (PIL)). In Simulink, each of these models is automatically translated into the next more concrete model. Embedded Tester is then used to verify environment assumptions made in the more abstract model (performance for example) and the generation mechanism itself.

[1] See `http://www.osc-es.de/index.php?idcat=17`

Another point to consider is that if the model for test generation is so complete that one can generate code out of it, it is in most cases so complex that there is no difference between model-based testing and directly testing the SUT.

The second scenario is concerned with automatically extracting a model from code after the system was built manually based only on some specification. This leads to the same problem of not having any redundancy between intended and actual behavior. What can be tested using this approach is again only whether extraction of the model and the necessary abstraction work correctly.

The third scenario is the first that generally provides the required redundancy since the model is manually created based only on the specification and requirements. This makes it possible to automatically assign verdicts, i.e. decide whether the test passes or fails, thereby verifying the code.
In this approach, it is also possible that the activities of creating the specification and building the system are not performed by the same organization as it is often the case for example in automative industry.

Finally, the scenario that uses two distinct models for code generation and test generation is clearly the one with the highest effort required. But this could pay off since it combines both the advantages of model-based testing and model-based development. In this scenario, it has to be assured that the two models are different from each other. This requires at least that they are created by seperate people.

Blackbox and Whitebox Models

Models can furthermore be differentiated in blackbox and whitebox. Blackbox means in this context that the underlying structure of the SUT is not known and has to be explored during testing. Only the state the system is currently in, is visible. When in contrast considering whitebox models, the structure of the system is known completely. Note the difference to blackbox or whitebox *testing*. Model-based testing is generally blackbox testing as it tests against requirements, even though one might argue that it sometimes employes structural information about the software.

When comparing both, whitebox models have some obvious advantages. Because the whole model is known, paths through the model can be planned in advance. This allows the use of more sophisticated algorithms and also to make a statement about coverage. The downside is that a tool that generates test cases out of a whitebox model has to understand the semantics of the declarative modeling language.

Blackbox models however are not limited to declarative languages like state machines or UML. Their only constraint is that their actions have to be executable. This makes it possible to use familiar programming languages with all their features as a modeling language. Thus, blackbox models are often easier to write and for a tool to execute.

This paper focuses on whitebox models, since they currently are the more popular approach.

Models applicable in test case generation
A classic model for model-based testing is a **finite state machine (FSM)**. A FSM consists of states, transitions between them, and actions that are performed at a given moment. It is called finite because its set of states is finite. FSMs are generally well-suited for model-based testing, because the model trace generation can be reduced to well-known and established graph-traversal algorithms like random walk or chinese postman (all transitions are visited as soon as possible).

But the downside of FSMs is that the model gets very large for a real system. Because of this problem of *state space explosion*, model-based testing switched largely to **extended finite state machines (EFSM)**. An EFSM allows state variables that form the data state. Additionally, it facilitates the use of guards and actions on transitions (cf. Figure 4). If the guards are fulfilled, the transition can fire, thereby bringing the machine in the next state and setting the variables accordingly. The advantage of an EFSM is that is has a small number of visible states but a large number of invisible states that are defined through the state variables. This is a kind of new abstraction layer that is put on simple FSMs. Through these additions, an EFSM is in most cases smaller than an FSM while allowing a more expressive model, especially when a real system is modeled. This makes EFSMs better-suited for test generation.

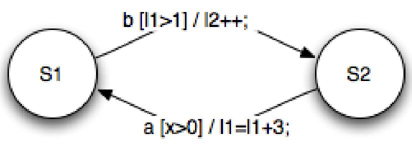


Fig. 4. A simple extended finite state machine (EFSM). Triggers `a` and `b` with corresponding guards. When for example transition `a` is fired and the guard is true, the local variable `l1` is incremented by `3`.

[8] provides more theoretical background of EFSMs and shows how functional vectors can be automatically generated out of them. These can then be used for test generation.

A **State chart** [9] is also an extension of an FSM targeted especially at reactive systems. It features hierarchies and concurrency. States can be expanded into lower-level state charts, which again add new levels of abstraction. This improves understandability and reduces the state space. A rather unique feature is that state charts can be explicitly modeled to be executed in parallel which is hardly possible using (E)FSMs. A state chart features optionally also triggers (a signal or event that fires a transition), guards (a boolean condition that when false prevents the transition from being fired) and actions (what is performed if the transition is fired). State charts may be more understandable than FSMs, but are not trivial to work with.

State charts were also adopted in the **Unified Modeling Language (UML)**[2] where they are called state machines. They also feature parallelism and hierarchies. Transitions are also described using triggers, guards and actions. The main benefit of using UML state machines instead of state charts or EFSMs, is its integration with other model artifacts, e.g. a class diagram. Operations that were defined in a class diagram are available as triggers in a state machine. Commercial tool support may also be better for UML. Actions are usually described using the *Object Constraint Language (OCL).*
A subset of UML useful for model-based testing, called UML-MBT, is presented in [10]. UML-MBT is based on three diagrams: class diagrams (model points of control and observation of the SUT), object diagrams (define test data) and state machines (model dynamic behavior). It is complemented by a subset of OCL that is used for transition actions and postconditions.

Labelled transition systems (LTS) [11] are similar to finite state machines, with the exception that neither their state space nor their number of transitions has to be finite. Labelled referes to the fact that their transitions have a label that has some meaning depending on the semantics of the concrete model. One prominent example in this context is the Input Output Transition System (IOTS) [12], which is the basis of several model-based testing tools.

State based or pre/post notations model a system as a collection of variables that represent the internal state of the system, and operations that modify them. Operations are usually defined not using programming languages but a set of pre and post conditions. Examples of these languages are B and Z.

There are many other modeling languages applicable for model-based testing. This includes Markov chains that allow statistical testing, decision tables/trees, grammars, and message sequence charts (MSCs).

Choosing a model

As there is no silver bullet model that fits all purposes, choosing a model is a trade-off between different properties. The first criteria is the system under test. The domain of this system may eliminate some models and favor others up-front. For example, if the system is dealing with accepting inputs following a particular structure, grammars may be a better choice over finite state machines. If parallelism is needed, state charts may be the best choice.

Other points to consider are skills, audience and tools. People creating and maintaining these models must be experienced enough to do this in reasonable time and accuracy. Tool support that helps during this tasks must be available and has to be evaluated. Important is also the audience of the models: do they have to be understandable only by an automatic test generator or also by the stakeholders on the customer side?

Organizational issues are also to be considered. For example, it may be relevant at which phase in the development cycle the models are created. In an earlier stage they may be more abstract than in a later stage. The development process, for example waterfall versus iterative approaches, is also relevant here.

[2] A notation by the Object Management Group. `http://www.omg.org`

2.3 Test Specifications

In most model-based testing scenarios, a rather large, or even infinite[3], set of tests can possibly be generated. This set of test cases is often too large to be useful. Consider for example an embedded system that has to be reset after each test, which can take up to one minute. Clearly, not the whole test case set of maybe 3000 possible tests can be applied to that system as it would take days. Another problem is more general: tests that fail have to be interpreted by test engineers to find the cause. Even though model-based testing provides integrated traceability up to the requirements (cf. section 2.1), this can be a tedious task when confronted with a huge number of tests.

Test specifications offer the solution to this dilemma. They act as a selection criterion on the set of test cases, that is they "filter out" interesting test cases according to a predefined criteria. As with models, there is no *best* criteria; it has to be determined by a test engineer based on the actual situation.

[3] differentiates six categories of test specifications:

1. **Structural Model Coverage Criteria** exploit the structure of the model, e.g. the nodes and edges in a transition-based model or pre and post-conditions in state-based models. This means that test cases are generated according to a structural criterion and that test cases are selected if they increase the overall coverage of this criterion.
   For transition-based models, there exists a variety of graph coverage criteria, e.g. *all nodes* or *all paths*. Additionally, *control flow* and *data flow* coverage criteria have been adapted from white-box testing.
   Control flow coverage is based on the notion of condition and decision. A condition is a boolean expression and a decision is a point in the specification where control flow can take different paths, e.g. an `IF-THEN-ELSE` construct in a programming language. Upon these terms, different coverage criteria have been defined: *decision coverage* requires that each possible outcome of every decision is produced at least once. *condition coverage* is the equivalent for conditions. But both of them have weaknesses, because each alone does not cover all possible decisions and conditions. To overcome these deficiency, *decision condition coverage* was defined that requires that each possible outcome of every condition in each decision is produced at least once *and* that each possible outcome of each decision is produced at least once. Decision condition coverage is for example applied in [13].
   Data flow coverage focuses on the way values are assigned to variables and how these assignments influence the further execution. Thus, they require test cases from a variable assignment to the point where this variable is used. One data flow coverage criterion is *all definitions coverage* which requires that all defined variables are tested once. A stronger criterion is *all uses criterion* that requires that all variables are tested in all their uses. But this does not cover that all possible paths to reach a variable use are tested. This is ensured by the *all definitions uses paths coverage*.

[3] Infinite sets of test cases can among others be caused by loops in the model.

A detailed view on control and data flow coverage criteria is out of scope of this paper. Please refer to [14] for more information.

2. **Data Coverage Criteria** aim at choosing few test values from a large possible input data set. This can be done by splitting the set into equivalence classes, so that hopefully all data in one class is equivalent in terms of failure detection. It is then possible to test only one representative of each class. This approach is often accompanied by boundary analysis, which means testing the data at the border of each class.
3. **Requirements-Based Coverage Criteria** is applicable if model elements can directly be related to requirements. It is also possible to use so called scenarios, which specify an expected usage of the system under test. This can for example be concrete instances of UML Use Cases. Test generation can then be restricted to the specified scenario thereby reducing the number of test cases.
4. **Ad-hoc Test Case Specification.** A test engineer can use explicit test case specifications to control the test case generation. These specifications can be written in the same language as the model (but they do not have to as long as they are formal) and can then restrict the paths in the model that should be tested or can focus the test generation on critical parts. Those parts could for example be heavily used, safety-critical or single points of failure.
5. **Random and Stochastic Criteria** are mostly applicable to environment models. They result from analysis of user behavior and system usage and can then be used to apply usage patterns to the SUT. The simplest case is that all system functions have equal probabilities, which results in random selection of test cases. However, when there are functions that are more frequently used or are more critical, they should be tested more thoroughly.
6. **Fault-based Criteria** measure the test case's ability to detect faults in the software. The most common approach are mutation tests that change something in the implementation of the SUT and test if the generated test cases find these intentional errors.

Coverage criteria can also be useful to judge the adequacy of the selected test cases, e.g. to reason whether testing is complete. Thus, it is important to measure coverage between the test suite and the model/specification even if the test specification is not based on coverage.

Empirical Evidence

A summary of empirical evaluations comparing structural coverage-based testing with random testing is presented in [14]. According to these results there is no evidence that coverage-based testing performs significantly better than the much cheaper random testing. Random testing is also more likely to gain a higher overall coverage. The main advantage of coverage-based approaches is that they explore the SUT more systematically thereby finding more specific faults like for example bad treatment of bounds. The authors suggest the combination of random and coverage-based testing so that on the one hand coverage-based testing finds specific faults while random testing on the other hand provides confidence about reliability.

However, another evaluation [15] that compared coverage-based testing with random testing shows different results. For the experiment, a total of 24 errors were intentionally applied to a moderate-size software system in advance. Then, different test suites were generated and compared in terms of failure-detection. The authors describe that random testing found only 62.5%, whereas the coverage-based approach found up to 83.3%.

2.4 Test Generation

The problem of test case generation boils down to finding test cases that match the predefined test specifications. How test cases are generated depends again heavily on the modeling language used. *Theorem proving* can be used to generate test cases from formal specifications in pre/post notation, e.g. B or Z. Finite state machines can be analyzed by *model checking* or graph algorithms. Another technique that could be used is *symbolic execution* which can be applied to different kinds of models.

Theorem Proving

The basic idea of using theorem proving is that the input data can be partitioned into a smaller number of equivalence classes, so that (hopefully) data from the same class causes the same error or no error in the case of success. This property is called a *uniformity hypothesis*. Each equivalence class refers to one test case. These equivalence classes are obtained by trying to construct a formal proof for the model.

Unfortunately, theorem provers, especially in real application scenarios, are often only semi-automatic, because they have to be guided by a user. This makes it very time-consuming and expensive. However, there may be some quality requirements that can only be reasonably met via this costly procedure.

As the exact procedure of test case generation via theorem proving requires some knowledge about the specification language and is quite complex, the reader is referred to [16], which describes how test cases can be generated out of Z specifications. Prolog can also be used as a specification language thereby exploiting the integrated theorem proving capabilities.

One might argue, that if a formal proof is found that the model is correct, why should one construct additional test cases? On the other hand, if testing is unavoidable, why should one construct a formal proof in the first place? Besides the fact that proving the model's correctness does not necessarily make a statement about the SUT's correctness, the answer is that theorem provers perform a detailed analysis of the input domain in order to construct their proofs. The structure of the proof thus reflects the set of equivalence classes that fullfil the uniformity hypothesis, implicitly. Test cases can then be extracted out of the proof and applied to the SUT.

Model Checking

In model checking, the problem of test case generation is reduced to finding witnesses or counter-examples to a set of temporal logic formulas. These temporal logic formulas can be understood as coverage criteria, i.e. test case specifications. This allows to find a set of witnesses or counter-examples fully automated in a model that represent a set of test case specifications. The resulting execution traces (either the correct ones or the incorrect ones) can then be applied to the SUT.

Using model checking has the main advantage that all test generation logic is encapsulated in model checkers. Thus, the test engineers can fully focus on creating the model and test specifications in temporal logic. Model checking is also a quite mature technology that has been applied very often and successfully over the recent years.

The main downside is that model checking has problems when faced with state space explosion. It then becomes very slow and thus impossible to use. Therefore, attempts have been made to combine model checking and theorem proving to overcome the deficiencies of each other [17].

Graph Algorithms

For transition-based models, such as (extended) finite state machines, there exist a couple of graph algorithms that can be applied to the problem of test case generation. These algorithms traverse the graph that is defined by the FSM's states and transitions and thereby obtain model traces. Examples of these algorithms are: *random walk*, *chinese postman* and *all transition pairs*.

The random walk of length N is the simplest algorithm. It goes through the graph at random with a maximal number of steps N. Thus, it generates model traces of length N, but these traces can involve loops and testing some transition repeatedly while not at all including others. As mentioned earlier while discussing the coverage criteria, randomness in model-based testing may perform astonishingly well. Usage models can also be taken into account in the random walk algorithm simply by giving transitions with higher usage profile a higher probability, so that the algorithm more often follows these transitions.

Chinese postman [18] is a more sophisticated algorithm that covers all transitions in a minimal amount of time. It tries to avoid duplicate transitions.

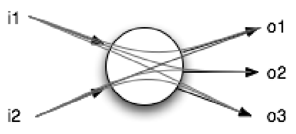


Fig. 5. *All transition-pairs* test generation algorithm.

All transition-pairs means that for each input transition to a state all transition outputs are followed. Figure 5 illustrates the algorithm. Both inputs `i1` and `i2` follow to each output `o1`, `o2` and `o3`. Offutt and Abdurazik [19] discuss different test generation techniques for UML-based specifications including all transition-pairs. They also offer a small empirical evaluation where transition-pairs performs better than a manually created test suite with 100% statement coverage.

Symbolic execution

Symbolic execution is similar to the informal technique that a test engineer applies a number of input values to the SUT and then compares the output to the expected output. The main difference is that in symbolic execution symbols instead of concrete values are applied, e.g. an x that represents the whole input domain of integer values.

The main goal of symbolic execution is then to explore the possible execution paths of the SUT and probably find errors in it. As a by-product test cases can be generated, because it is then known, which input values can force the SUT through particular control paths.

Lucio and Samer [16] describe approaches that first involve transforming the abstract model into a *constraint logic programming (CLP)* language. This program is then symbolically executed to find interesting model traces. Theorem proving is used as support for state space exploration. One symbolic trace then represents many concrete traces. This is why it has to be instantiated with concrete values before being applied to the SUT. This CLP approach is applied in the tool AutoFocus, which is described in Section 4.

2.5 Test Execution

The test execution can be differentiated into online and offline. Online means that each test is run directly after it has been generated. The model is tested on the fly. Offline refers to first generating all tests and then running them in a seperate step and maybe in a different environment.

Offline tests have the advantage that they can be run many times after they have been generated once, whereas online tests are regenerated every time. However, online tests have advantages at testing non-deterministic systems, because they can directly react to the system's responses.

2.6 Testing Non-Functional Requirements

It is important to note that model-based testing is not only useful for testing functional requirements. Non-functional requirements can also be verified.
Some of the requirements could be added to the model itself. This includes timing information, e.g. for real-time systems, which could be annotated to transitions. Other non-functional requirements can be tested through special test specifications. This way, it is for example possible to generate special stress test suites

which would include a high number of test cases, which could be executed in parallel, testing the most important functionalities of the system.

Before they can be tested, non-functional requirements have to be collected and specified formally. Refer for example to pages 1ff. for an overview of this topic.

However, testing non-functional requirements is not in the current mainstream of model-based testing and research is still in progress.

2.7 Example

Consider the following small example of a drink vending machine. The machine takes coins of values 50 and 100. One drink costs 200 coins. When the necessary amount is placed into the machine, additional coins should not be accepted. At any point, the user is able to press the return button to abort the operation and get back their money.

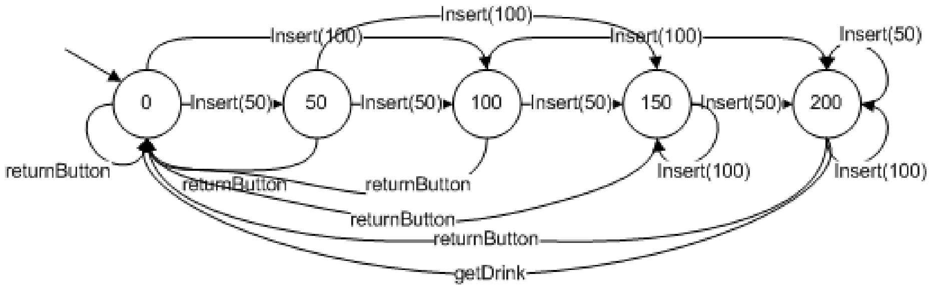


Fig. 6. FSM of the drink vending machine.

Figure 6 depicts the finite state machine of the drink vending machine. Out of this state machine, now a test suite could be generated using simple graph traversal. The model traces of a random test suite with a maximum length of 10 could for example look similar to this:

```
1. (0, insert(50), 50)
2. (50, insert(100), 150)
3. (150, insert(100), 150)
4. (150, insert(100), 150)
5. (150, returnButton, 0)
6. (0, returnButton, 0)
7. (0, insert(100), 100)
8. (100, insert(50), 150)
9. (150, insert(50), 200)
10. (200, returnButton, 0)
```

The transition coverage of this test suite would be $\frac{9}{16}$ or 56%. Due to random testing, the transition `(150, insert(100), 150)` gets executed twice in a row. A coverage-based approach would try to minimize these duplications and select the transitions that increase the overall coverage. In this case, it would have for example chosen the transition `(150, insert(50), 200)` instead. The transition `(200, getDrink, 0)` and therefore the SUT's operation `getDrink()` remains totally untested.

The generated model traces are then applied to a test execution framework, that is connected to the SUT via an individually developed test driver. The framework invokes the according operations on the SUT (`insert(x)`, `getDrink()` and `returnButton()`) and observes its behavior. It tests as long as either an error is found or the test suite is completed successfully.

3 Case Studies and Empirical Evaluation

This section focuses on the practical relevance of model-based testing. It presents several case studies that are described in the literature, with focus on industrial applications, and discusses empirical data helping to evaluate the technique's practical performance.

Based on their analysis of eight different case studies in the fields of processors, protocols and smart card testing, Prenninger *et al.* [20] conclude with the following observations:

1. The use of model-based testing is motivated by the complexity of today's systems. It is getting very hard to validate system's against their requirements using traditional (manual) approaches.
2. All case studies follow a common abstract process[4]. This has the advantage that the process is systematic, repeatable and measurable.
3. Model-based testing starts to be broadly applied in the domain of processor verification. The existence and wide-spread use of VHDL and alike languages providing well-defined abstraction levels, allows partial automation of the test model creation process.

In their conclusion, Prenninger *et al.* also criticize the lack of rigorous assessments including cost/benefit relationships when comparing model-based testing with traditional testing techniques and other quality assurance techniques. By now there exist many reports about the practical use of model-based testing, but there are few studies that really measure model-based against random or manual testing. One evalution that focuses on these questions, was conducted in [13] and will be described in the following.

The authors used the case study of an automotive infotainment network controller to evaluate different testing techniques in terms of error detection, model coverage and implementation coverage. In particular they compared test suites that were automatically generated and those that were manually created.

[4] The process is identical to the one described in this paper, mainly in section 2.

In both cases it was differentiated between using a model for test generation and not using a model. Additionally, the relation between explicit functional test selection criteria and pure random test selection was evaluated. In total, seven different test suites were created, which are depicted in Table 1.

Suite	Automation	model	TC specs	Comment
A	manual	yes	yes	
B	auto	yes	yes	
C	auto	yes	no	random creation with model
D	auto	no	n/a	random creation without model
E	manual	no	n/a	only requirement MSCs
F	manual	no	n/a	additional MSCs
G	manual	no	n/a	traditional techniques

Table 1. Different test suites used in [13]

Based on informal *message sequence charts (MSC)* that were included in the requirement specification, the network controller was modeled in the *AutoFocus* CASE tool using EFSMs. In test suite *F*, additional MSCs were created while clarifying the requirements. For the automated test case generation, the model was then transferred into a *constraint logic programming (CLP)* language and the test specifications were added. This CLP was then executed, thereby enumerating all traces of the model[5]. In addition to the number and type of detected errors, the condition/decision (C/D) coverage both between $testcases \leftrightarrow model$ and $testcases \leftrightarrow implementation$ was measured.

Figure 7 shows some of the empirical data that was collected. It can be seen that the use of models significantly increases the number of detected requirement errors. However, the number of detected programming errors seems not to depend on the use of a model. Pure random tests (*C* and *D*) detect fewer errors than all other test suites. Different test suites detect different errors while no suite detects all errors. This suggests that the combination of different approaches is promising.
Concerning coverage, it can be infered that model and implementation coverage correlate moderately. Implementation coverage cannot exceed a level of 75% due to abstractions applied to the model. Coverage and error detection overall correlate positively. However, there is no evidence that greater coverage leads to more detected errors. This questions the validity of using coverage as test selection or adequacy criterion.
With the same number of tests, handcrafted and automated model-based tests found an equal number of errors. A sixfold increase in the number of generated tests found an additional 11% of errors.

[5] Actually, the CLP was symbolically executed for test cases with a maximum length of 25. Technologies of test case generation were described earlier in Section 2.4

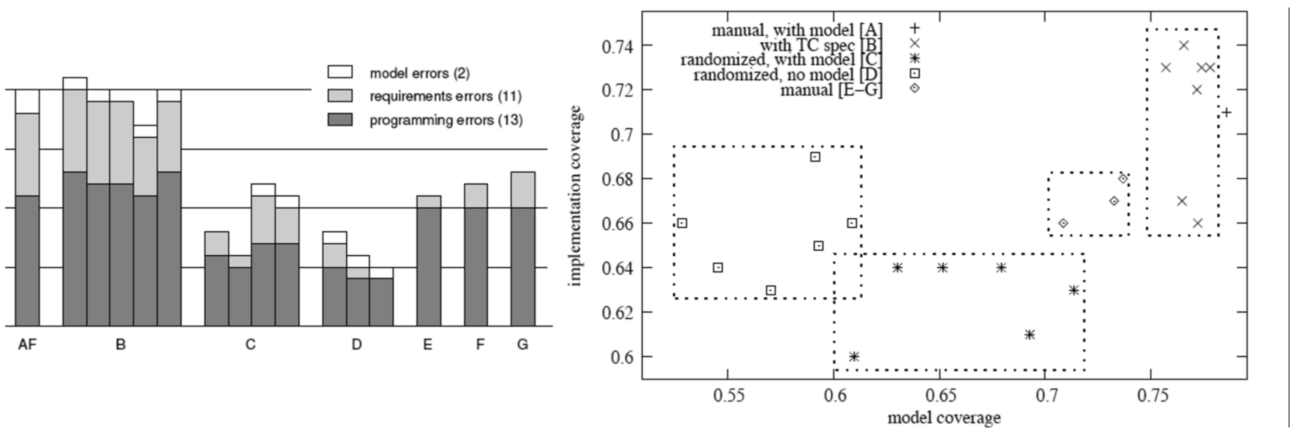


Fig. 7. Detected errors and coverages of the different test suites (from [13])

The high number of remaining errors in the MSC-based requirement documents suggests that they should be complemented by the model itself.

Note that the authors of the case study do not make an explicit statement about the economics of model-based testing and its automation, i.e. whether the additional modeling time pays off in the end. This is related to the concept of Return On Investment (ROI), which is popular in the business world. Giombetti (pages 65ff.) gives an overview on how to measure cost/benefit of software quality assurance.

Another case study reports about a model-based testing tool called *Spec Explorer* [21] that was developed at Microsoft Research. According to the authors, the tool is used on a daily basis by about 100 testers. In a comparison between testing with Spec Explorer and testing traditionally, Spec Explorer helped to discover 10 times more errors than the traditional approach while costs for modeling and for traditional test automation were roughly the same. Especially, deep system-level bugs were better found by the model-based testing tool. Unfortunately, the study does not go into greater detail clarifying its claims.

A case study that was already briefly mentioned above, was conducted in [19]. It is about a small experiment that the authors conducted in order to evaluate the usefullness of a tool they developed, which is able to automatically generate test case out of UML state charts. For this purpose, they compared three different scenarios: automatic test generation with *full predicate coverage* (a coverage method that takes the predicates on transitions into account and is similar to condition/decision coverage), automatic generation using *all transition pairs*, and manual creation by a test engineer. For each technique, one test suite was created and applied to a moderate-size software product that had been seeded with errors in advance. Their results state that *full predicate coverage* found all 25 errors, followed by *transition pairs* which found 18/25 and thus performed significantly better than the manual test suite with 100% statement coverage that found 16/25 failures.

A recent study [22] was aimed at giving an overview of the empirical evidence concerning model-based testing. It categorizes 85 papers according to modeling language and type of evidence, e.g. speculation, experience, experiment, etc. A special focus was also on the use of UML as modeling language. Only 7 studies were describing actual experimental results - the vast majority (72%) was only speculation or included an example without giving evaluation criteria on which a comparison would be possible. The experiments were conducted on either toy systems or small applications up to 6500 LOC. In most cases, engineers applied intentional defects to the system and tried to find them by testing. The results show that 100% of defects can be found using a large test case set. But also small test sets have a good coverage (about 70-90%). The tradeoff between larger and smaller sets, e.g. in terms of runtime and memory footprints, is still not very well researched. All these results conclude that there is still no strong evidence about the practical usefulness of model-based testing regarding complexity, cost, effort, and required skill.

4 Tool Support

This section aims at examining if and how much the preceding theoretical foundations have found their way into practice. Model-based testing, especially when automatic generation of test cases is included in the definition, is not possible without appropriate testing tools. In this section, some tools that are available are presented briefly trying to give an overview and to establish the connection to the preceeding chapters. Due to commercial restrictions this was not always possible. Then, a practical example using one tool is described to give the reader an insight in how model-based testing is done in practice today.

4.1 Tool Descriptions

This section does not claim completeness but rather aims at giving a small overview. For a more complete review of model-based testing tools, please refer for example to [23].

AutoFocus

AutoFocus [24] is a tool developed at the Technical University of Munich[6] targeted at the graphical modeling and development of distributed systems. The core element of AutoFocus models are components that can be connected via typed ports. These networks of components are depicted by *system structure diagrams* (SSD). SSDs can be hierarchical, i.e. components can be decomposed in other communicating sub-components. While these diagrams describe the static structure of the software system,EFSMs are used to specify the dynamic behavior

[6] See `http://autofocus.in.tum.de`

of each component. AutoFocus components are time-synchronous, which means that a global clock is used and all components act simultaneously. Thus, an AutoFocus model is essentially a set of time-synchronous EFSMs communicating over typed channels.

In addition to the model, test specifications and the SUT itself are needed. Test specifications can either be functional, structural or stochastic. The test generation process is very similar to the one described in Section 2.4. The model is first transformed into a CLP program, which is enriched with the test specification. The predicates of low-level components K in the program are of the form

$$step_K(S_K, i, o, D_K) \Leftarrow guard(i, D_K) \land assgmt(o, D_K)$$

This means that if the guard is true, the transition with input i from S_K to D_K can be fired thereby outputting o and modifying the component's data state according to the assignment. The predicates for higher-level components, i.e. components that are composed of other components, can be translated recursively.
Then, the program is symbolically executed to find the possible model traces. As a further optimization, states are not visited multiple times. Before they can be applied to the SUT, the resulting symbolic traces have to instantiated. This can happen either by random or by limit analysis.

Further details on the test generation procedure as well as a case study can be found in [6].

AGEDIS

The AGEDIS[7] (automated generation and execution of test suites for distributed component-based software) project [25] was a cooperation between several industrial and academic research groups under leadership of IBM funded by the European Commission that ended in 2004.

First, an open architecture was defined that allows for flexible and interchangeable use of concrete components. Existing tools from different research groups can be adapted to the interfaces defined and make use of the AGEDIS infrastructure.

The AGEDIS architecture is depicted in Figure 8. It defines the following six interfaces between the different components, while the first three are user interfaces and the last three are internal interfaces:

– **Behavioural modeling language**. AGEDIS uses AML, which is a profile of the UML, as modeling language. However, because of the flexible architecture, support for other modeling languages is claimed to be also possible. Class diagrams describe the structure of the SUT while state machines are used to model the behavior of each class. Actions are described using Verimag's language IF. Stereotypes define the interface between the SUT and its environment. Additionally, an object diagram describes the initial state of the system.

[7] See `http://www.agedis.de`

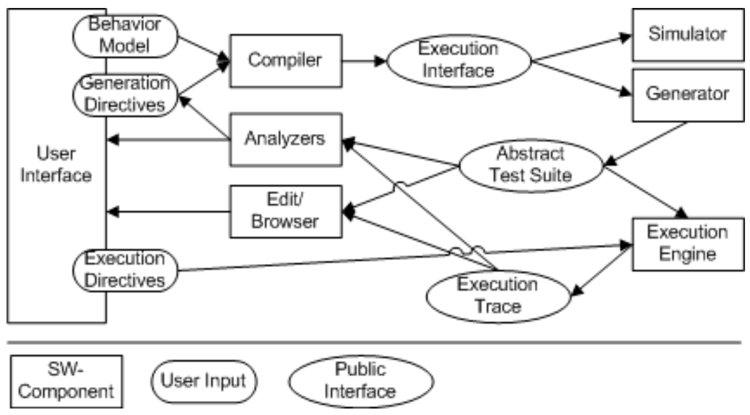


Fig. 8. The AGEDIS architecture (from [25])

- **Test Generation Directives** are similar to Test Specifications. They instruct the test generator how to generate test cases. The directives can be modeled as MSCs or system-level state diagrams. Since they are often not easy to create, AGEDIS provides a set of predefined directives including random generation, state coverage and transition coverage.
- **Test Execution Directives** instruct the test execution where and how to execute the tests. They provide a mapping between the abstract datatypes of the model and the real datatypes of the SUT. It additionally contains configuration values for the test execution (host information for distributed systems, delays, timeouts, etc.). The test execution directives are described in XML.
- **Model Execution Interface**. This interface is responsible for the encoding of the behavioral specification, which consists of classes, objects, and state machines. Again, IF is used to do so.
- **Abstract Test Suite** and **Execution Trace**. Both abstract test suite and test suite traces are described using the same XML schema, which again allows for interoperability. A suite consists of one or more test cases and traces and a description of the SUT model. One test case contains a set of test steps which in turn may consist of stimuli, observations, and directions for continuation to the next step of for reaching a verdict. Several actions may take place in one step, either in parallel or sequential.
  Test cases can be parameterized so that one test case can be run multiple times with different parameters. Also, one test case can invoke other test cases as sub-steps.

Modeling in AGEDIS is possible using the Objecteering UML Editor with an AML profile. AGEDIS also features a simulator for the constructed IF model that shows how the model reacts to specified inputs. This should enable the user to see their model in action and understand it more fully.

Test generation requires translating the model to a *labeled transition system* (LTS). This LTS is traversed from the initial state to a state marked in the test generation directives as “Accept”, i.e. a state that reaches an accept verdict, and a set of test cases is extracted. The algorithm is based on the TGV engine, which is derived from model checking, and further described in [26].

A test execution framework named Spider was also developed. It provides platform and language independent utilities for test case execution, logging and tracing of distributed systems. Spider is able work with SUTs written in C, C++ and Java. It takes care of the distribution of objects and controls the whole test run. It generates stimuli, makes observations, compares them to the indented behavior, and writes the executed traces in an XML format for further analysis.

The test analysis framework consists of two tools: a coverage analyzer and a defect analyzer. The coverage analyzer is able to generate test purposes that instruct the test generator to generate test cases that cover missing values or sequences in the model. The defect analyzer clusters the traces that lead to the same fault and tries to generate a single trace out of them. This should ease the manual analysis when there are many repeated occurences of the same fault.

Several case studies to evaluate the AGEDIS tools were also published [27]. The case studies showed mixed results. While the industrial testers were pleased with the test execution framework and the overall interoperability of the AGEDIS tools, they were critical with the modeling language and the test generator. The used UML state machines were not always seen natural. The use of the IF language was criticized and the AGEDIS authors admit that they probably should have used OCL or a subset of Java instead. The test generator seemed not to be scalable enough for large industrial applications.

AGEDIS is available free of charge for academic purposes. The small german company imbus AG[8] was charged with developing commercial applications of the project.

ModelJUnit

ModelJUnit[9] is a very simple open source tool developed by Mark Utting at the University of Waikato. As the name intends it aims at extending JUnit with model-based testing capabilities.

Models for ModelJUnit are (E)FSMs that are written as Java classes. It is possible to generate tests for these models using various test generation algorithms (mainly based on graph traversal) and measure coverage figures.
The following listing shows a simple FSM expressed in ModelJUnit.

```
public class FSM implements FsmModel {
  private int state = 0;  // 0..1
  public FSM() { state = 0; }
  public String getState() { return String.valueOf(state); }
  public void reset(boolean testing) { state = 0; }
```

[8] See http://www.imbus.de

[9] See http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

```
  public boolean action0Guard() { return state == 1; }
  public @Action void action0() { state = 0; }

  public boolean action1Guard() { return state == 0; }
  public @Action void action1() { state = 1; }

  public boolean actionNoneGuard() { return state != 1; }
  public @Action void actionNone() { /*leave state the same.*/ }
}
```

The java class is equal to the graphical notation depicted in Figure 9.

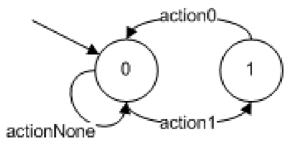


Fig. 9. Above FSM in graphical notation.

The model class does also act as the test driver that connects the model to the SUT. The methods annotated as *@Action* typically include code to call the SUT and check if the results conform to the expected results.

The approach of using a blackbox model has the downside that the test generator can only build up the model at runtime by checking which guards are true, but has no knowledge about the internal structure of the model. When there are guards that are rarely true, it is difficult to obtain the whole model. Another liability is that the communication with the SUT is handled internally without explicit modeling. This leads to limitations in the test generation algorithm and the coverage measures. Additional information on ModelJUnit can be found in [2].

Conformiq Qtronic

Qtronic [28] is a commercial tool developed by Conformiq Software[10]. A Qtronic model consists of up to three parts:

– **Textual source files** written in the modeling language QML, which is a superset of Java that describes data types, constants, classes and methods. The additions to Java include macros, nondeterministic programming, guards and requirements traceability.
– **UML Statecharts** that model the dynamic behavior of the classes.
– **UML Class diagrams** to declare classes and their relationships

[10] See `http://www.conformiq.com`

The graphical models (Statecharts and class diagrams) can be created in various commercial UML tools or in the Qtronic Modeler, but are completely optional. All behavior and static structure of the system can also be described directly in the textual source files.

The interfaces to the SUT are expressed as a collection of *ports*. The test driver that connects model and SUT and handles abstraction/concretization is called *test harness*. Unfortunately, details about the test generation algorithms are not disclosed to the public. It is only stated that they "employ a combination of advanced technologies [...] (including e.g. symbolic execution)" [28].

As test specifications, Qtronic features a number of criteria, including requirements coverage (requirements are annotated in the model), transition coverage and all paths coverage. QTronic aims at constructing a minimal test suite that conforms to the test specifications to ease test execution and later debugging activities.

It is also possible to include timing information in the test cases, i.e. timings for messages sent to the SUT and received from it.

Qtronic is available as a 30-day trial version. In the next section, a practial model-based testing example is given using Qtronic.

4.2 An Example

The same example of a drink vending machine is used that was already described in Section 2.7. First, a model of the system is created. This is done in combination of a textual description in QML and a graphical using UML state machines. Figure 10 shows an excerpt of the state machine in Qtronic Modeler.

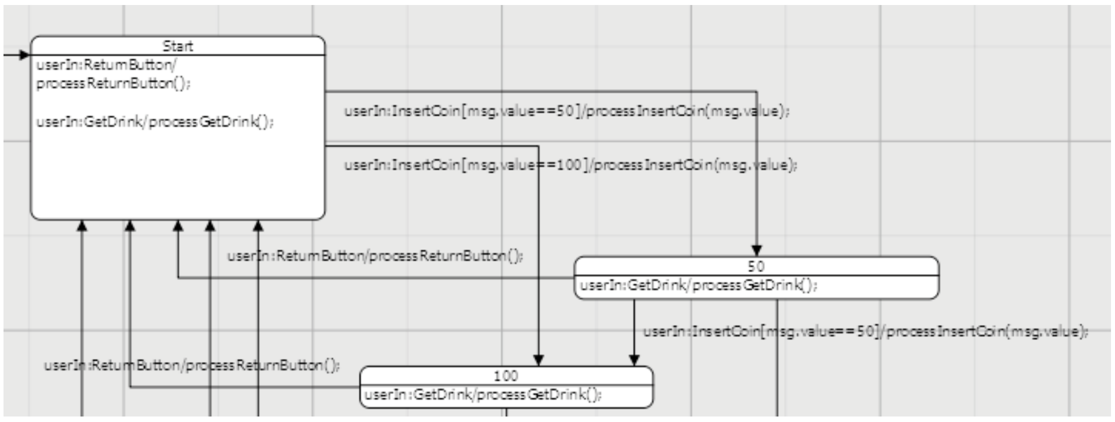

Start
userIn:ReturnButton/ processReturnButton();
userIn:GetDrink/processGetDrink();
userIn:InsertCoin[msg.value==50]/processInsertCoin(msg.value);
userIn:InsertCoin[msg.value==100]/processInsertCoin(msg.value);
userIn:ReturnButton/processReturnButton();
50
userIn:GetDrink/processGetDrink();
userIn:InsertCoin[msg.value==50]/processInsertCoin(msg.value);
userIn:ReturnButton/processReturnButton();
100
userIn:GetDrink/processGetDrink();

Fig. 10. Excerpt of the state machine in Qtronic Modeler.

```
private void processInsertCoin(int value)
        if (value == 50 && balance == 200 ||
                value == 100 && balance == 150) {
                // ignore
        } else {
                balance += value;
        }
        CurrentBalance curBal;
        curBal.balance = balance;
        userOut.send(curBal);
        requirement "Coins can be inserted";
}
```

At the beginning it is checked whether additional coins are still valid or can be ignored because a balance of 200 would be exceeded. Then a message is sent to the environment that contains the current balance. The last line denotes that the requirement named "coins can be inserted" is realized in this method. This is useful for traceability reasons.

When the model is finished, the Qtronic tool is able to generate test cases out of it. Figure 11 shows an excerpt of one generated test case that is visualized as an MSC.

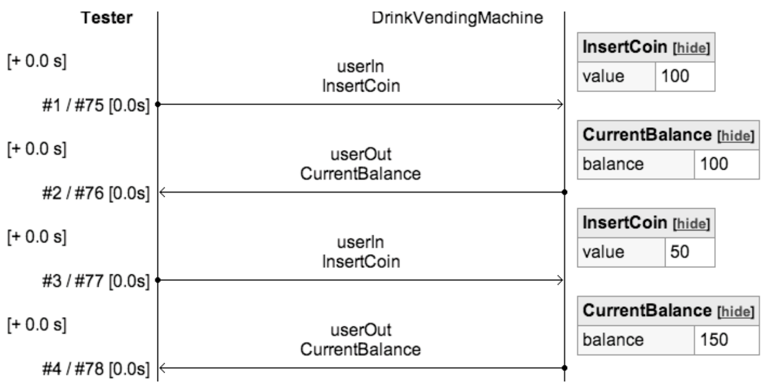


Fig. 11. Excerpt of one test case generated by Qtronic.

The MSC shows the communication between environment and SUT including the content of the messages. In total, a number of 52 test cases were created in 22 seconds that yield an overall transition coverage of 100%. Test case can also be generated into several executable formats, so that they can directly be executed without further manual actions necessary.

Qtronic also supports the online execution of test cases while they are created. For this purpose, an adapter has to be created that mediates between the Tester and the SUT. For the sake of simplicity, this is not described in this section.

While Qtronic features an easy-to-use graphical user interface that may also be usable by project managers or other people not so concerned with technical things, testing with it is still not trivial. The model (and SUT adapter) creation is the crucial part in the workflow. Learning the QML notation requires some effort and the documentation is lacking some important points.

4.3 Choosing a Tool

As with models, there is no tool that can be used uniformly for all testing purposes. The choice of a tool depends heavily on the domain of the software system under test, e.g. whether it is an embedded system or an enterprise information system. Additionally, skills and experience of the test engineers have to be considered since the presented tools use different modeling languages as well as test generation routines. Prices and licensing might be another consideration.

Some comparisons of model-based testing tools have already been conducted. Please refer to [23] for a summary and for hints to the original literature.

5 Conclusion and Outlook

This paper gave an overview of model-based testing in theory and practice. It started with explaining the theoretical foundations behind the technique. Since there is not *the one* model-based testing process, possible variations in the different steps were also discussed. This included the different modeling languages applicable, different test generation algorithms and test specifications including some empirical evidence. The fact that there are multiple valid approaches, which may all perform well at a specific project, makes clear that good test engineers are still needed. Only their scope shifts from manually designing test cases to finding the best modeling language and choosing the test generation algorithm including the best test specifications for the concrete software system and project.

In the next section the practical relevance of model-based testing was examined. Several case studies and experiments that aimed at evaluating the technique were presented. The results of the studies was sometimes contradictory: some studies were rather sceptic, while others were very positive. All in all, more research, especially cost/benefit and complexity related studies, are needed.

Section 4 gave an overview of academic as well as commercial tools for model-based testing. It was tried to establish the connection to theory where possible. The section concluded with a practical example that gave an insight how practical tools are used today. It showed that model-based testing is still no push-button technology, since creating appropriate models is far from being trivial even for such a small example system.

Despite about 30 years of research, the high number of publications on the topic and a number of existing tools, model-based testing is still no completely established technique in today's software projects. It has to be questioned critically why this is not the case. Today, mostly small and medium size embedded systems or even only the critical parts of it, are tested model-based. Model-based testing for large systems, like e.g. enterprise information systems, is still a research topic. It might be that the technique is still too complex to apply, especially in larger projects. The theoretical foundations are difficult and have to be hidden as far as possible from the user without limiting his abilities. The contradicting and incomplete results from case studies might be another hindering point as there are only few companies that are willing to experiment in their important projects. And last, companies may not see the benefit of creating yet another model when their traditional testing approach seems to work, too.

But, since software systems become more and more complex, the need for model-based testing might increase over time, since creating and analyzing test cases manually then becomes too complex, error-prone and time-consuming. With software becoming more ubiquitous, the role of software quality becomes even more important. When model driven development is further applied in companies this might strengthen the role of models in the development process and in the heads of the responsibles, and thereby act as a catalyzer for model-based testing. However, it has to be considered that model driven development is also in competition with model-based testing. If a company creates a model of their system, they are, because of the necessary model redundancy, only able to either generate code or test cases out of it. In the case of code generation, model-based testing can only be used to verify the code generation mechanism itself and the environment assumptions made. In practice, especially in embedded systems development, this scenario is quite common, where model-based testing is only used to check the automated translation between the different abstraction stages, e.g. software in the loop and hardware in the loop. An alternative would be to only create test models for the most critical parts of the SUT.

Having considered this, it is still very interesting how model-based testing will develop in future. All in all, chances are high that the attractive attributes of model-based testing, like being repeatable, coping with changing requirements, conserving knowledge, and last but not least the benefits of the mere modeling activity, are put into wider practical use.

References

1. Pretschner, A., Phillips, J.: Methodological issues in model-based testing. Lecture notes in computer science **3472** (2005) 281–291
2. Utting, M., Legeard, B.: Practical model-based testing. Morgan Kaufmann (2007)
3. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. Department of Computer Science, The University of Waikato, New Zealand, Tech. Rep (2006)
4. Stachowiak, H.: Allgemeine Modelltheorie. Springer Wien (1973)

5. Ludewig, J.: Models in software engineering–an introduction. Software and Systems Modeling **2**(1) (2003) 5–14
6. Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., Scholl, K.: Model-based test case generation for smart cards. Electronic Notes in Theoretical Computer Science **80** (2003) 170–184
7. Prenninger, W., Pretschner, A.: Abstractions for model-based testing. Electronic Notes in Theoretical Computer Science **116** (2005) 59–71
8. Cheng, K., Krishnakumar, A.: Automatic functional test generation using the extended finite state machine model. Proceedings of the 30th international on Design automation conference (1993) 86–91
9. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (1987) 231–274
10. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise uml for model-based testing. In: A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing, New York, NY, USA, ACM (2007) 95–104
11. Keller, R.: Formal verification of parallel programs. Communications of the ACM **19**(7) (1976) 371–384
12. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software - Concepts and Tools **17**(3) (1996) 103–120
13. Pretschner, A., Prenninger, W., Wagner, S., Kuehnel, C., Baumgartner, M., Sostawa, B., Zoelch, R., Stauner, T.: One evaluation of model-based testing and its automation. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM (2005) 392–401
14. Gaston, C., Seifert, D.: Evaluating coverage based testing. Lecture notes in computer science **3472** (2005) 293–322
15. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Software Testing Verification and Reliability **13**(1) (2003) 25–53
16. Lucio, L., Samer, M.: Technology of test-case generation. Lecture notes in computer science **3472** (2005) 323–354
17. Rajan, S., Shankar, N., Srivas, M.: An integration of model-checking with automated proof checking. Computer-Aided Verification **95** (1995) 84–97
18. Thimbleby, H.: The directed Chinese Postman Problem. Software Practice and Experience **33**(11) (2003) 1081–1096
19. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. Proc. Second International Conference on the Unified Modeling Language (1999) 76
20. Prenninger, W., El-Ramly, M., Horstmann, M.: Case studies. Lecture notes in computer science **3472** (2005) 439–461
21. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with spec explorer. Formal Methods **3582** (2005) 542–547
22. Neto, A., Subramanyan, R., Vieira, M., Travassos, G., Shull, F.: Improving Evidence about Software Technologies: A Look at Model-Based Testing. Software, IEEE **25**(3) (2008) 10–13
23. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for test case generation. Lecture notes in computer science **3472** (2005) 391–438
24. Huber, F., Schatz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems. Industrial Applications and Strengthened Foundations of Formal Methods, LNCS **1313** (1997)

25. Hartman, A., Nagin, K.: Model driven testing-AGEDIS architecture interfaces and tools. In: Proc. 1st European Conference on Model Driven Software Engineering. (2004) 1–11
26. Jeron, T., Morel, P.: Test generation derived from model-checking. Computer Aided Verification **99** (1999) 108–122
27. Craggs, I., Sardis, M., Heuillard, T.: AGEDIS Case Studies: Model-Based Testing in Industry. In: Proc. 1st European Conference on Model Driven Software Engineering. (2004)
28. Conformiq: Conformiq qtronic semantics and algorithms. Technical whitepaper, Conformiq Software, http://www.conformiq.com/downloads/ConformiqQtronicSemanticsAndAlgorithms.pdf (4 2008)

Cost/Benefit-Aspects of Software Quality Assurance

Marc Giombetti

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
giombett@in.tum.de

Abstract. Along with the ever more apparent importance and criticality of software systems for modern societies, arises the urgent need to deal efficiently with the quality assurance of these systems. Even though the necessity of investments into software quality should not be underestimated, it seems economically unwise to invest seemingly random amounts of money into quality assurance. The precise prediction of the costs and benefits of various software quality assurance techniques within a particular project allows for economically sound decision-making.
This paper presents the cost estimation models COCOMO, its successor COCOMO II and COUALMO, which is a quality estimation model and has been derived from COCOMO II. Furthermore an analytical idealized model of defect detection techniques is presented. It provides a range of metrics: the return on investment rate (ROI) of software quality assurance for example. The method of ROI calculation is exemplified in this paper.
In conclusion an overview on the debate concerning quality and cost ascertaining in general will be given. Although today there are a number of techniques to verify the cost-effectiveness of quality assurance, the results are thus far often unsatisfactory. Since all known models make heavy use of empirically gained data, it is very important to question their results judiciously and avoid misreadings.

1 Introduction

The usage of software is pervasive in our society and software has taken a central role in our daily business and private life. Software is used in planes, trains, cars, banking systems aso., and therefore the software's quality plays a crucial role. The quality is important for the acceptance of the software by the user and thus is a key factor to the success of the software product.

Software systems are expensive products because their construction involves a lot of skilled people. Companies which develop software often spend excessive amounts of money to get an high quality software, which overcomes the firms actual quality needs. On the other hand some companies do not take quality assurance seriously enough or do not spent enough money, or do not use the right techniques for the quality assurance of their software production. It has often been seen that companies let an immature software skip over to field production. A possible software failure may then lead to millions of breakdown costs, loss of reputation, loss of market shares or even injure people. Thus, it is important to find the right balance between quality and quality assurance costs. The available budget should be invested pareto-optimally into the right quality assurance techniques to get the appropriate quality given a certain budget.

This work focuses on the main questions of how software quality assurance can be applied economically. It will give an insight into software quality cost, its calculation and present some models which enable the selection of the appropriate amount of specific quality assurance techniques to find the best solution for the investment in quality assurance effort. An idealized model of defect-detection techniques will be presented and this model can be used as starting point to calculate different metrics as return on investment.

1.1 Software quality

With respect to software system quality, it is not always possible to achieve the "best quality", but the intension is to create a software system having the right quality for a given purpose.

As a matter of fact, it is important for each software development project to define its specific meaning of software quality during the planing phase. On the one hand the quality of a software is adequate if all the functional requirements are met. On the other hand the softwares quality is also defined over non-functional requirements as reliability, usability, efficiency, maintainability and portability. This set of characteristics is defined in the international standard for the evaluation of software product quality ISO/IEC 9126-1:2001 [1]. For each characteristic there exists a set of sub-characteristics which all contribute to the software quality to some extend. A more detailed description of the ISO 9126 can be found in the pages 1ff.

As example one could look at the availability of software. For an office application, an availability of 99,9%, which corresponds to an average downtime of 8,76 hours/year, is fairly appropriate. In respect to availability this office application is of high quality. Contrary to this, a power plant control software, having the same availability of 99,9% which stands for an average downtime of 8,76 hours/year too, is definately not acceptable. An unsafe failure might result in a disaster, polluting the environment and possibly injuring people. This shows that it is not enough only to consider a certain metric of a quality characteristic, but that it is important to see the application environment of the software too. Additionally it is important to look at financial issues. Quality assurance is costly and the expenses to be made to achieve the right quality are of interest to the project management and the customer. It is necessary to estimate and measure software quality costs.

1.2 Software quality costs

We have introduced different types of software quality characteristics and mentioned that it is not always possible to achieve the best quality, but that it is important to get the right quality for a certain software. The reason why the achievement of the "best" software quality is not possible, is mainly a financial issue. The higher the software's quality, the higher the quality assurance costs. Unfortunately the relation between the software quality improvement and the quality assurance investment effort is not linear. Todays software engineers and project managers are more and more aware of the software costs incurring over the entire software lifecycle. It is of paramount importance to find the right trade off between the software development quality assurance costs and the possible costs which arise when the software fails at the client. This is the only way of veraciously handling all the costs of a software system.

In the last three decades, there has been a lot of scientific work on finding relationships between quality and cost of quality. *Quality costs* are the costs associated with preventing, finding and correcting defective work [2]. The *Cost of Quality* approach is such a technique. Mainly it is an accounting technique that is useful to enable the understanding of the economic trade-offs involved in delivering good-quality software. Commonly used in manufacturing, its adaptation to software offers the promise of preventing poor quality [3]. But which relationships exist between quality and cost of quality?

According to J.M. Juran and F.M. Grynas *Juran's Quality Control Handbook* [4] as well as P. Crosbys book *Quality Is Free* [5], the cost of quality can be subdivided into two categories: *Conformance* and *nonconformance* costs. Figure 1 gives an overview on the quality and how it relates to different types of costs.

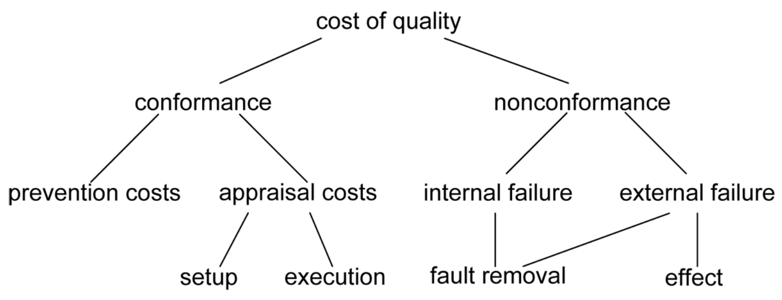


Fig. 1. Overview over the costs related to quality [4] - Extension of S. Wagner [2]

Conformance costs, also known as control costs, can be partitioned into *prevention costs* and *appraisal costs.* Prevention costs include money spend on quality assurance, so that the software meets its quality requirements. Prevention costs for example are tasks like training, code and requirements reviews, tool costs and quality audits. All these quality assurance activities *prevent* the injection of various types of faults. The appraisal costs emerge from activities like developing test cases, reviews, test data and executing the test cases. *Setup* and *execution* costs are the most common appraisal costs. Setup costs cover all initial costs for configuring the development environment, acquiring licenses for test tools, aso. The execution costs cover all the costs which arise during the actual test-runs, review meetings aso. The execution costs are mainly personnel costs.

Nonconformance costs, also known as *failure of control* costs, emerge when the software does not meet the defined quality requirements. Nonconformance costs exist in two forms: *Internal failure* costs and *external failure* costs. The first type contains all the costs that arise due to a misfunction of the software during development and before the final delivery to the customer. The second type contains the costs that result from a failure of the software at the customer. After the establishment of a test process the internal failure costs increase and at the same time the external failure costs go down. There is an interdependency between both types of costs.

The nonconformance costs contain *fault removal* and *effect* costs. The fault removal costs are linked to the internal failure, as well as the external failure costs. This means that removal costs arise if a failure is detected internally as well as if the software fails at the customer. The last important type of costs are the *effect* costs. Effect costs arise when the software fails externally. They include all the costs that are caused by the software failing at the customer. Failure costs are not part of the effect costs. Examples for effect costs are loss of sales because of reputation, law costs, loss adjustment payments aso. [2]

Up to now we have seen which types of quality costs exist and we will proceed by taking a look at how quality and software costs in general, can be estimated.

2 Quality and cost estimation

Cost estimation models are not new to the software industry, and today there exists a whole set of cost estimation models. Cost estimation is important because it removes some of the uncertainty in respect to the required expenditures. The objective is to make the best possible estimation of the costs, given a set of project factors and the skills of the development team. On the one hand companies need good cost estimates to be competitive in the market and to win call for bids. On the other hand it is important for them not to underestimate costs, because if they offer projects for a fixed price, they will narrow profits or even lose money. One of the first software cost estimation models has been developed by Barry Boehm in the 1970s. In his book *Software Engineering Economics* [6] Boehm presents the COnstructive COst MOdel (COCOMO).

Quality estimation is a bit more complex and requires more advanced models. Whereas costs can be measured, it is more difficult to measure quality. The

COnstructive QUALity MOdel COQUALMO is an extension to COCOMO and aims at determinating the software quality. Because the quality of a software product is directly related to the number of residual defects in the software, COCOMO takes the approach of predicting the quality of the software by estimating the the number of residual defects per/KDSI (Thousand of Source Lines of Code).

2.1 COCOMO & COCOMO II

In the following we introduce COCOMO as well as COCOMO II and provide some information on these techniques because they constitute the basis for COQUALMO. COQUALMO is a software quality estimation model and this work will mainly focus on this technique as an example for a quality estimation model. COQUALMO will be presented in detail in Section 2.2.

The main feature of COCOMO is to predict the required effort for a software project. Boehm developed COCOMO empirically by running a study of 63 software development projects and statistically analyzing their their project characteristics, people skills, performance and invested effort. The output of the COCOMO model is an effort prediction for the software development expressed in months. Because the main software development cost driver is the developer activity and the resulting personnel costs, one can assume that cost and effort are nearly the same.

To predict the effort the following equation is used:

$$\text{EFFORT} = a \cdot (\text{KDSI})^b$$

The constants a and b vary according to the type of project. KDSI is a measure to determine the software size, namely the Kilo Delivered Source Instructions. COCOMO distinguishes between three different development modes to enable more accurate predictions:

- *Organic:* Projects are small and similar to previous projects and are developed in a stable and familiar environment.
- *Semi-detached:* Between organic and embedded mode.
- *Embedded:* Projects have tight an inflexible requirements and constraints and require a lot of innovation.

Table 1 contains the effort equations for the different development modes. As intuitively expected: with increasing complexity of the project, the parameters a and b increase and thus the required effort increases too.

We have now seen COCOMO in its most basic form: *the basic model.* There also exists an *intermediate model* and a *detailed model* which use an *effort adjustment factor (EAF)* which is multiplied with the effort calculation to get more accurate results. The EAF is the product of 15 cost factors subdivided into four categories: *platform costs*, *product costs*, *personnel cost* and *project costs.* In the intermediate model the following equation is used to determine the effort:

$$\text{EFFORT} = a \cdot \text{EAF} \cdot (\text{KDSI})^b$$

Development mode:	organic	$\text{EFFORT} = 2.4 \cdot (\text{KDSI})^{1.05}$
	semi-detached	$\text{EFFORT} = 3.0 \cdot (\text{KDSI})^{1.12}$
	embedded	$\text{EFFORT} = 3.6 \cdot (\text{KDSI})^{1.20}$

Table 1. COCOMO effort equations for different development modes.

We will not go into further detail on how to use COCOMO, because the first version is outdated and a lot of improvements have been made meanwhile. The interested reader may refer to *Software Engineering Economics* [6] for additional information on the first version of COCOMO.

The second version of COCOMO was developed in the 1990s and is mainly an adjustment of the first version to the modern development lifecycles and to the new techniques in software development. COCOMO II has been calibrated using a broader set of empirically collected project data, and in contrast to COCOMO additionally focuses on issues as:

– Non-sequential and rapid-development process models.
– Reuse driven approaches involving COTS packages, reengineering, application composition and application generation capabilities.
– Object oriented approaches supported by distributed middleware.
– Software process maturity effects and process-driven quality estimation.

Furthermore, COCOMO II now contains a set of new cost drivers. These cost drivers are subdivided into four categories: platform, product personnel and project cost drivers. Table 2 contains a list of these cost drivers, which are important and will be used as *defect introduction drivers* in COQUALMO in Section 2.2.

Both COCOMO and COCOMO II models make use of empirically collected data and are only as good as the accuracy of this data. The quality of the empirical data used to calibrate the model has a direct influence on the quality of the estimation outcome of the model. Estimations by definition tend to be subjective and should always be looked upon with the necessary skepticism. For example the constants a and b are not fixed by the model, but every software company should adjust them based on the experience they gain from their daily software projects. To make the models as useful as possible, as much data as possible should be collected from projects and used to refine the model. A good way to store and learn from daily project data, is the application of the *Experience Factory* approach proposed by Basili, Caldiera and Rombach [8].

COCOMO as well as COCOMO II do not make predictions on the quality of the software, neither which quality assurance techniques should be used to get the right quality at *optimized* costs. Quality is directly related to the number of defects which reside in the software. Intuitively, the more defects there are in the software, the poorer quality is. Therefore it is of interest to have a model which

Category	Cost driver
Platform	Required Software Reliability (RELY) Data Base Size (DATA) Required Reusability (RUSE) Documentation Match to Life-Cycle Needs (DOCU) Product Complexity (CPLX)
Product	Execution Time Constraint (TIME) Main Storage Constraint (STOR) Platform Volatility (PVOL)
Personnel	Analyst Capability (ACAP) Programmer Capability (PCAP) Applications Experience (AEXP) Platform Experience (PEXP) Language and Tool Experience (LTEX) Personnel Continuity (PCON)
Project	Use of Software Tools (TOOL) Multisite Development (SITE) Required Development Schedule (SCED) Disciplined Methods (DISC) Precedentedness (PREC) Architecture/Risk Resolution (RESL) Team Cohesion (TEAM) Process Maturity (PMAT)

Table 2. COCOMO II cost drivers [7].

quantifies the number of defects that get into the software as well as the number of defects that are removed from the software. COQUALMO is one instance of such a model and will be presented in the following:

2.2 COQUALMO

The COnstructive QUALity MOdel COQUALMO is an extension to the COCOMO II model. It determines the rates at which software requirements, design, and code defects are introduced into a software as a function of calibrated baseline rates, modified by multipliers determined from the projects COCOMO II product, platform, people and project attribute ratings [9]. It enables 'what-if' analyzes that demonstrate the impact of various defect removal techniques and the effects of these attributes on software quality. It additionally provides insights into determining shipment time, assessment of payoffs for quality investments and understanding of interactions amongst quality strategies. Additionally it relates cost, schedule and quality of software. These characteristics are highly correlated factors in software development and form three sides of the same triangle. Beyond a certain (the "Quality is Free" point [5]), it is difficult to increase the quality without increasing either the cost or schedule or both for the software under development [7].

With the development of COQUALMO Boehm aimed at facilitating the finding of a balance between cost, schedule and quality. Additionally to COCOMO II, COQUALMO is also based on the *The Software Defect Introduction and Removal Model.* The idea behind this model is that defects conceptually flow into a holding tank through various defect source pipes. Basically this means that the defects made during the requirements analysis, the design, the coding aso., flow through the defect source pipes into the software. On the other side there are also defect removal pipes through which the defects removed by quality assurance activities (a.e. testing) conceptually flow out of the software again. In the following the *Defect Introduction Model* and the *Defect Removal Model* will be presented and the relations which exist between COCOMO and COCOMO II will be outlined.

Defect Introduction Model: In COQUALMO, defects are classified based on the origin they result from. There exist *requirements defects*, *design defects* and *code defects.* The purpose of the Defect Introduction (DI) model is to determine the number of non-trivial requirements design and coding defects introduced into the software during development. As input to the DI model an estimation of the software size is necessary. This estimation may be thousand source lines of code (KDSI) and/or function points. Furthermore COQUALMO requires 21 (Disciplined Methods DISC is left out) of the 22 multiplicative DI-drivers of COCOMO II (see Table 2) as input data. The usage of the COCOMO II drivers not only makes the integration of COCOMO II into COQUALMO straight forward, but also simplifies the data collection activity and the model calibration which have already been setup for COCOMO II.

There exist three categories of severity of defects: *critical*, *high* and *medium.* To actually calculate the total number of defects introduced into the software the following formula is used:

$$\text{Number of defects introduced} = \sum_{j=1}^{3} A_j (\text{Size})^{B_j} * \prod_{i=1}^{21} (\text{DI driver})_{ij}$$

where j identifies the three artifact types (requirements, design, code) and A is a multiplicative constant which is determined experimentally. *Size* is the project size in KDSI or FP. B is initially set to 1 and accounts for economics of scale. Further details on the calibration of the model and the usage of parameter B can be found in *Modeling Software Defect Introduction and Removal: COQUALMO* [7]. Now that we can quantify the defects that get into the software, we look at the model of how they get out again.

Defect Removal Model: The Defect Removal (DR) model is a post-processor to the DI model. The concept main feature of DR model is to estimate the number of defects removed from the software by certain quality assurance activities.

These activities include three profiles: *Automated Analysis*, *People Reviews* and *Execution Testing and Tools*. Each of these profiles has different levels of increasing defect removal effectiveness from *very low* to *very high*. Table 3 contains the necessary defect removal investment and the rating scales for the mentioned profiles. The rating scales are the rows of the matrix whereas the activity profiles form the columns. The table then imposes what has to be done in each profile to achieve a certain rating level of quality. Additionally to every level of each of these profile, *Defect Removal Fractions* (DRF) are associated. These fractions are numerical estimates and have been determined by experts in a two-round Delphi estimation session. The interested reader can find information on the Delphi estimation process in [10].

Rating	**Automated analysis**	**Peer reviews**	**Execution testing and tools**
Very low	Simple compiler syntax checking	No peer review	No testing
Low	Basic compiler capabilities	Ad hoc informal walkthroughs	Ad hoc testing and debugging
Nominal	Compiler extension Basic requirements and design consistency	Well-defined sequence of preparation, review, and minimal follow-up	Basic test, test data management, problem tracking support; Test criteria based on checklists
High	Intermediate-level module and inter-module; Simple requirements and design	Formal review roles with well-trained participants, basic checklists, and follow-up	Well-defined test sequence tailored to organization; Basic test-coverage tools and test support system; Basic test process management
Very high	More elaborate requirements and design; Basic distributed-processing and temporal analysis, model checking and symbolic execution	Basic review checklists and root-cause analysis; Formal follow-up using historical data on inspection rate, preparation rate, and fault density	More advanced test tools, test data preparation, basic test oracle support, distributed monitoring and analysis, and assertion checking; Metrics-based test process management
Extra high	Formalized specification and verification; Advanced distributed processing	Formal review roles and procedures; Extensive review checklists and root-cause analysis. Continuous review-process improvement; Statistical process control	Highly advanced tools for test oracles, distributed monitoring and analysis, and assertion checking; Integration of automated analysis and test tools; Model-based test process management

Table 3. Defect-removal investment rating scales for COQUALMO [11].

As input the COQUALMO DR model requires the number of non-trivial *requirement, design and coding defects introduced* (= the output of the DI model). Furthermore the defect removal profile levels as well as the software size estimation are mandatory input parameters for the DR model. The model outputs the number of residual defects per KDSI (or per Function Point). The following example should give you a better feeling for the measure. Figure 2 shows a chart of the COQUALMO estimated delivered defect densities for the different defect removal rating categories. The chart is based on values of a calibrated baseline which have been rounded slightly to simplify the handling and to avoid an overfitting of the model. The rounded data contains 10 requirements defects, 20 design defects and 30 code defects for the *Very low* removal rating. One can see that for a *Very low* defect removal rating, 60 delivered defects are left in the software. If more effort is spent and a *Very high* rating level is achieved, the delivered defect density is reduced to 1,6 delivered defects per KDSI.

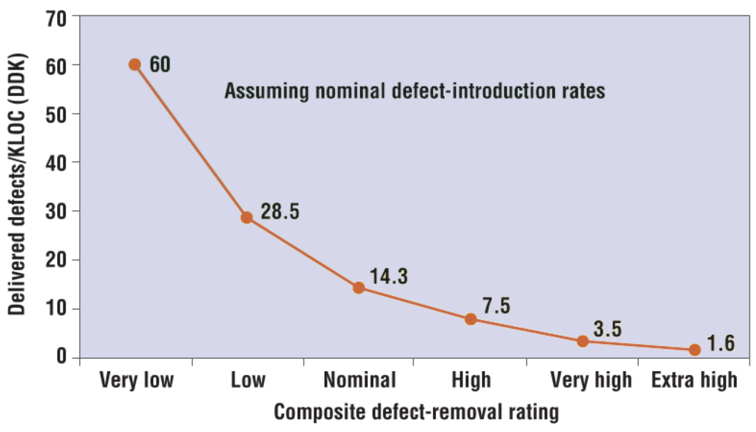


Fig. 2. Estimated delivered defect densities using COQUALMO [11]

The quality assurance team and the developers are mainly interested in the number of defects which reside in the software. The residual defects metric is important from a technical as well as a financial point of view, because every defect leading to a fault at the customer also leads to effect costs. The number of residual defects in artifact j is:

$$DRes_{Est,j} = C_j * DI_{Est,j} \prod_i (1 - DRF_{ij})$$

where C_j is a calibration constant, $DI_{Est,j}$ is the estimated number of defects of artifact type j introduced and i can take the values from 1 to 3 according to the type of DR profile (automated analysis, people reviews, execution testing and tools). The last variable DRF_{ij} is the Defect Removal Function for defect removal profile i and artifact type j. In the following we will highlight how COCOMO and COQUALMO are related.

Relationship between COCOMO II and COQUALMO: COQUALMO is integrated into COCOMO II and cannot be used without it. Figure 3 shows the

DI model and the DR model which are integrated into an existing COCOMO II cost, effort and schedule estimation. The dotted lines are the inputs and outputs of COCOMO II. Apart from the software size estimation and the platform, project, product and personnel attributes, the defect removal profile levels are necessary to predict the number of non-trivial residual requirements, design and code defects.

Because of this tight coupling between COCOMO II and COQUALMO, I think that a project manager should use COQUALMO for a software project if COCOMO II estimates already exists. The effort to implement COQUALMO is worth it, when considering the payoff a project can get from applying COQUALMO. Nevertheless there are also some drawbacks in the usage of COCOMO and COQUALMO. One big disadvantage of COCOMO is that is uses the Size (in KDSI) to calculate the effort. Because effort calculations are usually done in a very early project stage, when there is often not enough information to estimate the Size of the complete product. Also the weighting of the cost factors is not easy at this early point.

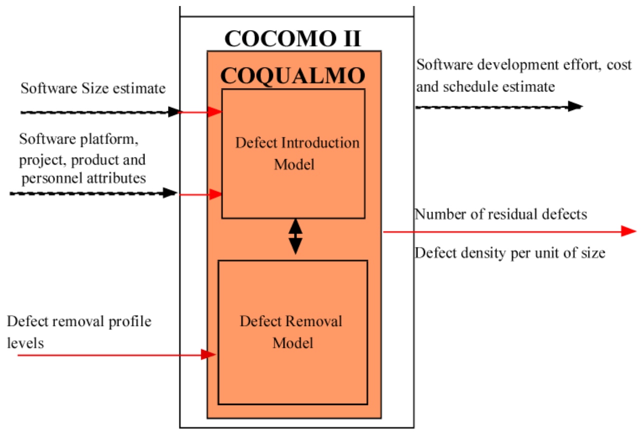

COCOMO II
COQUALMO
Software Size estimate
Software platform, project, product and personnel attributes
Defect removal profile levels
Defect Introduction Model
Defect Removal Model
Software development effort, cost and schedule estimate
Number of residual defects
Defect density per unit of size

Fig. 3. The Cost/Schedule/Quality Model: COQUALMO Integrated with COCOMO II [7]

Additionally the data which is collected using COQUALMO can again be used to improve the estimates of the different sizing parameters of COQUALMO and COCOMO II. Furthermore this data is the foundation of further investigations on software quality and cost. Section 2.4 illustrates an example on how return on investment calculations for software projects can be accomplished. The ROI calculations are one example of a metric which can be based on the analytical model of defect-detection techniques which will be presented in the following.

2.3 An analytical model of defect-detection techniques

The following analytical model of defect-detection techniques has been developed by Stefan Wagner as part of his PhD-thesis on *Cost-Optimisation of Analytical Software Quality Assurance* [12]. There are a few similar models, but this model was picked, because it clearly highlights the different cost components, precisely models the relations and is well documented. The model is a refinement and extension of the model by Collofello and Woodfield [13] which uses fewer input factors. It is a general model which can be used to analyze various types of quality assurance techniques. For example it can be used to analyze different types of testing or static analysis techniques to see which are most effective for a certain project. The model is a cost-based *ideal model of quality economics* and it doesn't focus on the use of the model in practice, but it is rather theoretical and mirrors the actual relationships as faithfully as possible.

Components: In the following we shall introduce the main components of the model, which is subdivided into three parts. All components are dependent on the spent effort t as global parameter.

– *Direct costs* $d(t)$, are costs which can be directly measured during the usage of a technique.
– *Future costs* $f(t)$, are the costs in the field which really incurred.
– *Revenues / saved costs* $r(t)$, are the costs in the field which could have emerged but which have been saved because failures were avoided due to quality assurance.

The model determines the expected values of these components, denoted by E. Before the introduction of the computation formulas, we will outline the model assumptions, to see which criteria have to be met to apply the model.

Assumptions: Because the model is an ideal model, the assumptions require an idealized environment:

– *The found faults are perfectly removed.* - This means that each fault which is found is removed and no new faults are introduced during the removal process.
– *The amount or duration of a technique can be freely varied.* - This is needed because there is a notion of time effort in the model to express for how long and with how many people a technique has been applied.

It is clear that in practice these assumptions cannot hold most of the time, but they are meant to clearly simplify the model.

Difficulty: The model requires a further notation for the characterization of quality assurance techniques. The difficulty of an application of technique A to find a specific fault i is denoted by $\theta_A(i)$. In a mathematical sense, this is the probability that technique A does not defect default i. Furthermore we introduce t_A which is the length of the technique application A. The length is the effort (a.e. measured in staff days) that was spent for the application of a technique.

In the later formulas, the expression $1-\theta_A(i, t_A)$ is the probability that a fault is at least detected once by technique A. Furthermore the model requires the concept of *defect classes* which regroups several defects according to the type of document they are contained in. For every defect there exists a document class (e.g. requirements defects or code defects), too. This subdivision in classes is important because some type of defect removal activity is only applicable to a certain type of defects. For example, it generally doesn't make sense to apply a formal verification technique to find requirements defects.

Defect propagation: Defect propagation is another important aspect which has to be taken into consideration. For the defects occurring during development, we know that they are not (always) independent of each other. One of the existing dependencies is the propagation of defects over the different phases of the software development process. The different phases are not considered, but the model rather takes the different artifact types and documents into consideration.

One defect in a certain artifact can lead to no, one or more defects in later documents. Figure 4 illustrates how defects may propagate over documents. In this example requirements defects lead to several design defects and a test specification defect. The design defects again propagate to code defects which can entail further test specification defects.

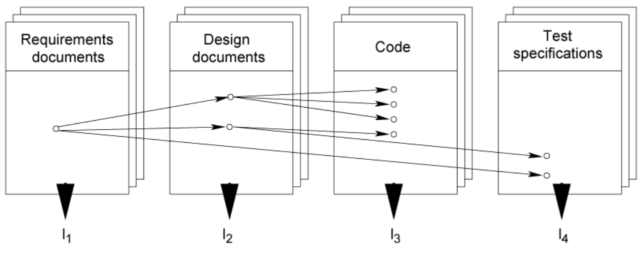


Fig. 4. Defect propagation over documents [12].

For each document type c, I_c is the set of defects in c and the total set of defects is $I = \bigcup I_c$. Additionally each defect has a set of predecessor defects named R_i, which may be the empty set. The usage of predecessors is important, because a defect can only be present in one artifact if none of the predecessors has already found it.

Model components: In the following the formulas for the expected values of the direct costs, the future costs and the revenues are given. Later the formulas for the combination of the different techniques will be given too. For the sake of simplicity, the defect propagation is not taken into consideration in the following equations. Nevertheless the defect propagation was introduced previously because it plays a main role in practice as well as in the ideal model. Presenting the defect propagation issue would go beyond the scope of this paper, therefore we will only introduce the simple equations. Details on the model can be found in [12].

Direct costs: *The direct costs are those costs that can be directly measured from the application of a defect-detection technique.* To determine the expected value for the direct costs $E[d_A(t_A)]$ the following formula is used:

$$E[d_A(t_A)] = u_A + e_A(t_A) + \sum_i (1 - \theta_a(i, t_A))v_A(i) \tag{1}$$

where u_A are the setup costs, $e_A(t_A)$ are the execution costs, $v_A(i)$ are the fault removal costs of the technique A and $1 - \theta_A(i, t_A)$ is the probability that a fault is at least detected once by technique A.

Future costs: The future costs are the costs which emerge in case some defects are not found. As introduced in Section 1.2, these costs can be subdivided into fault removal costs in the field $v_F(i)$ and failure effect costs $c_F(i)$. The equation to determine the expected value of future costs $E[f_A(t_A)]$ is the following:

$$E[f_A(t_A)] = \sum_i \pi_i \theta_A(i, t_A)(v_F(i) + c_F(i)) \tag{2}$$

where
$\pi_i = P(\text{fault i is activated by randomly selected input and is detected and fixed})$.

Revenues: Revenues are saved future costs. The cost categories are the same as for the future costs, but now we are looking at the defects that we find instead of the ones we miss. The equation to determine the expected value of revenues $E[r_A(t_A)]$ is the following:

$$E[r_A(t_A)] = \sum_i \pi_i(1 - \theta_A(i, t_A))(v_F(i) + c_F(i)) \tag{3}$$

Combination: up to now we have seen the equations for the different types of expected costs, which were always defined for one technique of quality assurance. Nevertheless, more than one technique to find defects is used in practice. The reason for this is that different techniques find different defects. The model takes this into consideration and allows the combination of different techniques. To formalize this, the model defines X as the ordered set of the applied defect detection techniques. To get the total direct costs, you have to sum over all technique applications X. Then we use Formula 1 and extend it that not only the probability that the technique finds the fault is taken into account, but also that the predecessor techniques have not found the fault. The predecessor defects R_i have to be taken into consideration too. To improve readability, we use

$$\Theta(x,i) = \prod_{y<x} \left[\theta_y(i,t_y) \prod_{j \in R_i} \theta_y(j,t_y) \right] \tag{4}$$

as the probability that a fault and its predecessors have not been found by previous (before x) applications of defect detection techniques. Then for each technique y that is applied before x, the difficulty for the fault i and all its predecessors in the set R_i is multiplied. The following formula is then used to determine the expected value of the combined direct costs d_X of a sequence of defect-detection technique applications X:

$$E[d_X(t_X)] = \sum_{x \in X} \left[u_x + e_x(t_x) + \sum_i ((1-\theta_x(i,t_x))\Theta(x,i))v_x(i) \right] \tag{5}$$

In the same way we can introduce the formula for the expected value of the revenues of several technique applications X:

$$E[r_X(t_X)] = \sum_{x \in X} \sum_i \left[(\pi_i(1-\theta_x(i,t_x))\,\Theta(x,i))(v_F(i)+c_F(i)) \right] \tag{6}$$

Here we consider the faults that actually occur, and that are detected by a technique. Additionally neither itself not its predecessors have been detected by previously applied techniques.

The expected value of the total future costs are the costs of each fault with the probability that the fault actually occurs and that all the techniques and the predecessors failed in detecting it. It can be determined using the following formula:

$$E[f_X(t_X)] = \sum_i \left[\pi_i \prod_{x \in X} \theta_x(i,t_x) \underbrace{\prod_{y<x} \prod_{j\ in R_i}}_{\Theta'(x,i)} \theta_y(j,t_y)(v_F(i)+c_F(i)) \right] \tag{7}$$

where $\Theta^{'}(x,i)$ is similar to Formula 4 but only describes the product of the difficulties of detecting the predecessors of i. In this case the probability if the predecessor has actually detected the fault is not necessary.

With the definition of the model components, it is now possible to calculate some economical metrics of the software quality assurance process.

Return On Investment: The return on investment ROI is a well known metric from economics. In economics the ROI is commonly defined as the gain divided by the used capital. In this case the rate of return for a specific defect-detection technique is of interest. We can calculate the ROI as follows:

$$\text{ROI} = \frac{\text{profit}}{\text{total cost}} = \frac{r_x - d_x - f_x}{d_x + f_x} \tag{8}$$

The profit is equal to the revenue minus the direct and future costs. The total cost is the sum of the direct and the future cost.

Return on investment can be an assessment of whether the investment in a specific defect-detection technique is justified by the quality improvement and the resulting cost reduction over the entire lifecycle. It sometimes even makes sense to have a negative ROI, when there is a very high risk to human life or the environment, or when there are important customer relationships which should not be jeopardised. It is even possible to express factors like loss of human life or the loss of a customer as costs. However this costs are often very difficult to quantify and in some cases legal issues may prevent this. It seems ethically not correct to express the loss of a human life because of a defect in a safety critical software system as effect costs. Nevertheless, theoretically there is no hindrance why it should not be possible to include such injury effort costs into the total costs of a software.

2.4 Example for a return on software quality investment calculation

In this section an example of how ROI calculations can be made in practice will be given. This example is part of the *The ROI of Software Dependability - The iDAVE Model* article by Barry Boehm et al. [9] and has been slightly adapted for this paper. The example focuses on two completely different software systems: The order-processing system of a mountain bike manufacturer and the NASA Planetary Rover. The intention is to determine the ROI of quality investments in availability for both system. Both systems have high availability requirements, but the project characteristics and the breakdown costs are completely different. In case of unavailability of the order-processing system, the mountain bike manufacturer will not be able to sell mountain bikes during the downtime. The unavailability of the NASA Planetary Rover would imply the end of its mission on a foreign planet, because it wouldn't be able to transmit it status to earth, neither be controllable from the command center anymore.

Return on investment calculations using iDAVE: The ROI analysis of this example is made using the Information Dependability Attribute Value Estimation (iDAVE) model. The iDAVE model is strongly related to COCOMO II and is a derivative of COQUALMO. We will not present the intricacies of this model, but we will focus on a practical example which does not require detailed knowledge on the iDAVE model. *"The iDAVE dependability ROI analysis begins by analyzing the effect of increasing dependability investments from the nominal business levels to the next higher levels of investment in analysis tool support, peer-reviews practices and test thoroughness"* [9]. The different levels of investment are the same as the defect removal levels in COQUALMO (see Table 3). The effects of investments are coupled to the RELY attribute, which is the COCOMO II reliability attribute (see Table 2). In the example the ROI calculations of the order-processing system will be done first, and the calculations for the NASA Planetary Rover will follow in the second part.

Mountain bikes order-processing system: The RELY rating scale for both the mountain bikes system includes the *Nominal*, *High*, *Very high* and *Extra high* ratings. For each project and the respective ranking levels the availability which is defined as:

$$\text{Availability} = \frac{\text{Mean Time Between Failure}}{\text{Mean Time Between Failure} + \text{Mean Time To Repair}}$$

is determined. The values of the *Mean Time Between Failure* and the *Mean Time To Repair* have been estimated by business domain experts. Table 4 gives the complete dataset for this example including the rating scales, availability calculations, financial values and the resulting return on investment rates.

Project	RELY rating	MTBF (hrs.)	MTTR (hrs.)	Availability	Loss (M$)	Increased value (M$)	Cost (M$)	Change (M$)	ROI
Mountain bikes order-processing system	Nominal	300	3	0,9901	5,31	0	3,45	0	-
	High	10K	3	0,9997	0,16	5,15	3,79	0,344	14,0
	Very High	300K	3	0,99999	0,005	0,155	4,34	0,55	-0,72
	Extra High	1M	3	1	0	0,005	5,38	1,04	-1,0
NASA Planetary Rover	High	10K	150	0,9852	4,44	0	22	0	-
	Very High	300K	150	0,9995	0,15	4,29	25,2	3,2	0,32
	Extra High	1M	150	0,99985	0,045	0,105	31,2	6	-0,98

Table 4. Values for the ROI calculation example [9].

For the order-processing system availability will be used as a proxy for dependability. Then it can be assumed that a one percent increase in downtime is equivalent to a one percent loss in sales. The total expected loss value for a given availability are now used to calculate the *Increased Value* for changing from one rating level to the next higher one. In the case of the order-processing system an improvement from *Nominal* to *High* RELY rating (from availability 0.9901 to $0,9997$) leads to an Increased Value (rounded values) of:

$$0,01 \cdot (531M\$) - 0,0003 \cdot (531M\$) = \\ 5,31M\$ - 160K\$ = 5.15M\$$$

where 531M$ is the arithmetic mean of the sales per year (the possible loss). With this, the dependability ROI calculation leads to:

$$\text{ROI} = \frac{\text{Profit}}{\text{Cost}} = \frac{\text{Benefits} - \text{Cost}}{\text{Cost}} = \frac{5,15 - 0,344}{0,344} = 14,0$$

A related result is that the additional dependability investments have relatively little payoff, because the company can only save $5,31 - 5,15 = 0,16M\$$ by increasing the availability.

The results of this analysis initially were discussed with business experts of the mountain bike order-processing quality assurance team who pointed out that a negative ROI that resulted from the improvement of a *High* to a *Very High* RELY level would not let their interest in improvements disappear. As when the needs in availability are fulfilled, there emerge other important motivators. Reducing security risks is an example for a possible motivator. This explains why negative ROIs are no reason to stop quality assurance investments.

NASA Planetary Rover: The NASA Planetary Rover has higher needs to availability as the order-processing system, and thus at least requires a *High* RELY rating level. The values for the *Mean Time Between Failure* are equal to those of the order-processing system, but the *Mean Time to Repair* increases. According to the engineers at NASA, it roughly takes a week or 150 hours to diagnose a problem, prepare the recovery and test its validity. The main requirement of the Rover is survivability. In short terms, survivability means that the Rover always has to keep enough power and communication capacity to transmit its status to earth. In this example availability is used as a proxy for survivability, because it is more straightforward to analyze. For the ROI calculation in this study, the total price of the mission (300M$) has to be known. The missions software costs account for 20M$, which is 7 percent of the entire mission cost. The software costs have been determined for the *Nominal* COCOMO II software RELY rating. Same as for the order-processing system one percent decrease in availability leads to a one percent of loss on the mission value of 300M$.

The dependability ROI analysis is yet again used to measure the effect of increasing the RELY rating to the next higher level. The transition from a *High* to a *Very high* RELY rating level corresponds to a cost increase from 22M$ to 25,2M$ and an increase in availability from 10K to 300K hours (see Table 4). The transition from a *Very High* to *Extra High* RELY rating is a special case because it exceeds the normal COCOMO II cost rating scale. It requires the extended COCOMO II RELY range. The required investment to fulfill this improvement is 6M$ and the added dependability will result in a negative payoff (ROI = -0,98). Nevertheless the NASA responsible pointed out that a negative ROI is acceptable for them too, because it reduces the risk of mission failure, loss of reputation or even harm to human life.

Example Summary: To sum up, Figure 5 compares the ROI analysis results of the mountain bike order-processing system and the NASA Planetary Rover. The chart summarizes Table 4 and points out how the return on investment rates change, while improving from one to the next higher level of reliability. It is clear that different projects have different break even points for the ROI rates. To conclude, there unfortunately is no recipe in when to stop quality assurance investments.

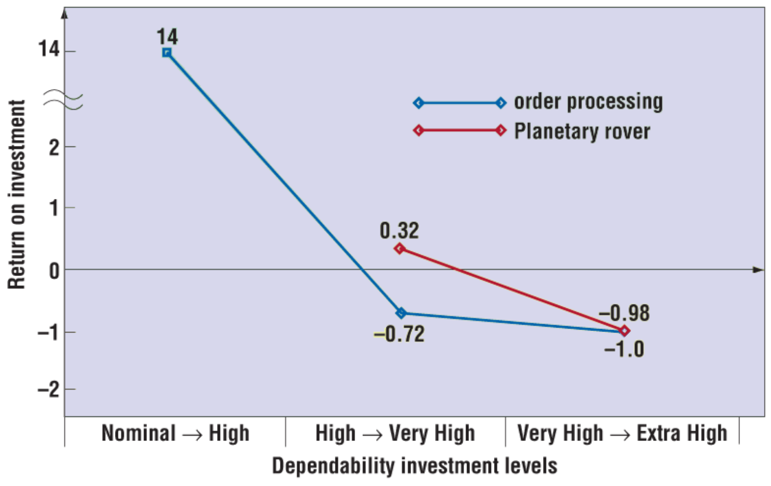


Fig. 5. Comparison of the ROI analysis for the mountain bike order-processing and the NASA's Planetary Rover [9].

3 Reflections on Cost-Effectiveness and Quality

In the following section I will make some deliberations on cost effectiveness and the quality of the presented models, as well as on the quality of the software product outcomes. This section reflects my personal opinion on those subjects. The first thing to be treated is the quality of the model calibration:

3.1 Model calibration

The different models that have been presented all make heavy use of empirical data. The problem with empirical data is that even though it might deliver good results for most of the application domains, it doesnt have to be appropriate for a specific domain. The model output directly depends on the quality of the empirical calibration data. The empirical data might be project data, as well as data collected from experts (expert opinion). There exist several statistical techniques to determine the quality of empirical data. An example on checking the validity of the calibration of models based on empirical data can be found in Freimut et al. [14]. A detailed study on different model calibration techniques was only mentioned for completeness reasons and would exceed the scope of this paper.

Nevertheless it is interesting to take a look at other problems related to the collection process and studies based on empirical data. One keyword in this context is *replicatability*. As already mentioned, results which are achieved in one specific software engineering domain, might be completely useless in another:

To illustrate how difficult the collection of empirical data is, and to see that even though the studies have been undertaken very precisely the results of two studies are completely different, it is interesting to take a look at *On the difficulty of replicating human subjects studies in software engineering* [15]. Even though this study is not about quality engineering, it shows how replications of the same experiences under equivalent conditions can lead to different implications. I picked this example because it is striking, well documented an reveals the difficulties of empirical software engineering and data collection.

The replication of results in software engineering is difficult, because it involves a lot of people having personal characteristics and a lot of influence factors on these people. In the original experience *The Camel Has Two Humps* undertaken by S. Dehnadi and R. Bornat, they claimed to have developed a test, administered before students were exposed to instructional programming material, that is able to accurately predict which students would succeed in an introductionary programming course and which would struggle. [16]. The authors of the paper set their sights in very carefully replicating a well documented experience to see if their results would agree or disagree with the original's. In either case, they would increase or decrease the confidence in the original hypothesis that *the camel has to humps*, so as to say that there are only the very good and very bad programmers, but no average programmers. Even though the replication was very precisely undertaken, the authors didn't achieve the same results as for the original experience. The authors tried to find out what could have lead to the different results, and they conclude that there are so many human and external factors, as that it is nearly impossible to replicate an experience and obtain the same results. For this paper, the previous replication experience shows that even though the model calibrations as well as the execution of the quality estimates might have been done very carefully, the models nevertheless may lead to untrustworthy results.

Same as for the model calibration, it is also interesting to reflect about quality-engineering concerns:

3.2 Quality-Engineering concerns

The models that were presented in this work are good to estimate some aspects of the software quality, which are easy to measure. Unfortunately most quality attributes are not easily measurable. The models for example do not consider the costs which result from a lack of maintainability or usability of the software. For the availability of a software system the case is "simple" because availability is an absolute measure. There is a relationship between increasing the quality assurance investments and the resulting availability increase which can be measured and validated.

Changeability: Software systems are subject to change. External influences as changing regulations, laws or simply changed market conditions require adaptations to the software. It the software system is unmaintainable for any reason, the emerging costs will be enormous, because the software has to be rewritten from scratch. It might even be worse: the software cannot be rewritten because the development would take far to long and exceed the companies budget. This scenario would be a complete disaster for the company and it might be entangled in lawsuits, because its software is involuntarily violating laws and regulations. To continue, I would also like to point out another interesting aspect to software, which is the *Total Cost of Ownership* in respect to quality.

Total Cost of Ownership: The presented models mainly took software aspects into consideration. Nevertheless the cost-effectiveness of a system also depends on hardware costs. Additionally power consumption in data centers to actually run the software cannot be neglected. In times of *Green IT* and increasing electricity fees, it is also important to include the power consumption by servers and cooling into calculation. One quality attribute of software (running on a server) could be its power consumption per hour. Therefore server-software is of good quality if it can be virtualized, thus reducing the number of servers, rack space, power consumption and costly management effort. This might sound awkward at first, but this could play an important role in the years to come.

Usability: On an even higher level, companies aiming at reaching more accurate ROI of software quality estimates should think about how many employees have to work with the software each and every day, and how frustrating the usage of the software might be. An effective, intuitive and easily usable software is of paramount importance. For example if several dialogs take to long to load, or are complicated to oversee, expensive working hours will be wasted for years each and every day. It is not neglectable that enterprise systems are expensive strategic investments, and thus will not seldom be used for a decade or even longer. As a matter of fact, different generations of employees will have to work with the software and will need to attend courses to learn how to handle it. The education costs are inversely related to the usability of the software. It is clear that if the software is to complex and cumbersome, the learning process slows down and takes longer, thus reducing the effectiveness of employees.

In my opinion it is important that in the future there should be more research and investigations to really capture all the influences software quality engineering has, thus enabling more precise rate of return calculations for software quality assurance.

4 Conclusion and outlook

The different cost estimation models presented in this work are the fundamental to the process minimizing, the total cost of software quality. But it is not sufficient that the software engineers are aware and know how to use such models. In contrary it is important to have a management team which is smart enough to look at the cost of quality over the entire lifecycle of the software product. It doesn't make sense to release immature software, just to reduce the short time expenses generated by quality assurance effort. The expenditure due to long term external failure in the field, will surely exceed the amount of saved short-term investments for quality assurance. Having an far-sighted and supportive management is of great necessity, but it is not a wild-card for achieving satisfactory return on software quality assurance investments. It is rather of paramount importance to know how to effectively make use of the presented models and always critically face the outcomes.

The presented models are a solid basis for companies to start their software quality assurance investments, but they are no push-button techniques. Software companies should always keep in mind the importance of the collection and appraisal of empirical data. They should see their companies as mature learning companies, and have robust experience factories where they save the data generated during the development process of their software projects [8]. The quality of the data in the experience base is crucial, because nearly all cost and quality estimation models make use of this data to predict cost and quality of future projects.

References

1. ISO/IEC: ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model. International Standards Organization, Geneva, Switzerland (2001)
2. Wagner, S.: A model and sensitivity analysis of the quality economics of defect-detection techniques. In: ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis, New York, NY, USA, ACM (2006) 73–84
3. Krasner, H.: Using the Cost of Quality Approach for Software. CrossTalk. The Journal of Defense Software Engineering **11**(11) (1998) 6–11
4. Juran, J., F.M., G.: Juran's Quality Control Handbook. McGraw-Hill (1988)
5. Crosby, P.: Quality Is Free: The Art of Making Quality Certain. Mentor Books (1980)
6. Boehm, B.: Software Engineering Economics. Prentice Hall PTR Upper Saddle River, NJ, USA (1981)
7. Chulani, S., Boehm, B.: Modeling software defect introduction and removal: Coqualmo (1999)
8. Basili, V., Caldiera, G., Rombach, H.: Experience Factory. Encyclopedia of Software Engineering **1** (1994) 469–476
9. Boehm, B., Huang, L., Jain, A., Madachy, R.: The roi of software dependability: The idave model. Software, IEEE **21**(3) (May-June 2004) 54–61
10. Stellman, A., Greene, J.: Applied Software Project Management. First edition edn. O'Reilly Media (2005)
11. Huang, L., Boehm, B.: How Much Software Quality Investment Is Enough: A Value-Based Approach. IEEE SOFTWARE (2006) 88–95
12. Wagner, S.: Cost-Optimisation of Analytical Software Quality Assurance. (2007)
13. Collofello, J., Woodfield, S.: Evaluating the effectiveness of reliability-assurance techniques. Journal of Systems and Software **9**(3) (1989) 191–195
14. Freimut, B., Briand, L., Vollei, F.: Determining inspection cost-effectiveness by combining project data and expert opinion. Software Engineering, IEEE Transactions on **31**(12) (Dec. 2005) 1074–1092
15. Lung, J., Aranda, J., Easterbrook, S.M., Wilson, G.V.: On the difficulty of replicating human subjects studies in software engineering (2008)
16. Dehnadi, S., Bornat, R.: The camel has two humps (working title). (2006)

Process Quality

Christina Katz

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
katz@in.tum.de

Abstract. In the past the quality of software was assured rather with a focus on the product quality, while nowadays the correlation of product quality and process quality carries more weight. This paper will give a short introduction to the quality standard ISO 9000 and the process improvement models CMMI and SPICE. Besides, a suitable method to measure process improvement using function points will be introduced and a discussion will show that improvement models are not yet perfect and probably will never be, but they should be given a chance.

1 Introduction

In the past the quality of software was assured with a focus on product quality. The process planning was a part of the structural and analytical quality assurance procedures.

In contrast to product quality, process quality describes the quality of the production process rather than the quality of the actual product. Hence, process quality is also related to development, production, management, organizational, and supply processes. Especially in software engineering there exists a correlation between process and software quality, which nowadays carries more weight. In the majority of software companies a self-contained process quality assurance is of particular importance.

In reference to software engineering process quality considers the software development process and is generally assured through the use of process models. A characterization and optimization of a company's process is often carried out using evaluation methods like CMMI or SPICE. In the paper on hand two process improvement models, CMMI and SPICE, and the quality standard ISO 9000 will be introduced and compared to each other. Besides that, a method to measure process quality using the function points metric is presented. Finally, the process improvement models are discussed.

2 Process Improvement Models and Quality Standards

Process improvement models provide useful frameworks for implementing and developing best practices and enhanced capabilities. When a company wants to improve its processes it is of avail to use a guiding model in order to be able to establish a structured procedure. In the following the quality standard ISO 9000 and the two process improvement models CMMI and SPICE, which can serve as a guiding framework, will be introduced.

2.1 ISO 9000 [1]

ISO 9000 is a family of standards for quality management systems and is maintained by ISO, the International Organization for Standardization, and administered by accreditation and certification bodies. The ISO 9000 family includes the standards ISO 9000 *Quality management systems - Fundamentals and vocabulary*, ISO 9001 *Quality management systems - Requirements*, and ISO 9004 *Quality management systems - Guidelines for performance improvements*. In the last years, ISO 9000 has become the most spread quality management model, since it can be set in in a wide range of businesses.

In this context ISO 9000 defines fundamentals, terms, and definitions for quality management systems. ISO 9001 defines the requirements on a quality management system, which have to be met when a company wants to provide products certified by ISO. It describes exemplary the entire quality management system and is a basis for Total Quality Management (TQM). ISO 9004 provides guidelines, which consider the efficiency of a quality management system. However, it forms no basis for certification or for a contract, but rather constitutes a management philosophy.

The standard ISO 9000 does not offer details about its application to specific domains of expertise, but there exists ISO 90003, which is a guideline for the software development industry. However, below the contents of ISO 9000 are described, since ISO 90003 does not change the demands of ISO 9000. It rather contains norms and guidelines to apply ISO 9000 to the development, delivery, and maintenance of software which have a more informative than normative character.

When following ISO 9000, a company or organization fulfills:

– the customer's quality requirements
– applicable regulatory requirements
– enhancement of customer satisfaction
– achievement of continual improvement of its performance in pursuit of these objectives

The quality management system standards of ISO 9000 are based upon eight quality management principles, which can be used as a guiding framework by the management. Those eight principles are defined in ISO 9000 *Quality management systems - Fundamentals and vocabulary*, and in ISO 9004 *Quality management systems - Guidelines for performance improvements*. The principles are:

Principle 1: Customer focus
Principle 2: Leadership
Principle 3: Involvement of people

Principle 4: Process approach
Principle 5: System approach to management
Principle 6: Continual improvement
Principle 7: Factual approach to decision making
Principle 8: Mutually beneficial supplier relationships

Below follows a short description of every principle, the benefits derived from their use and the actions that managers typically take in applying the principles to improve their organizations' performance.

Principle 1: Customer focus
Organizations depend on their customers and therefore should understand current and future customer needs, should meet customer requirements and strive to exceed customer expectations.

One key benefit of customer focus are increased revenues and market shares since responses to market opportunities can be done fast and flexible. Additionally, it leads to enhanced customer satisfaction, because the organization's resources can be used in a more effective way, and improved customer loyalty, which leads to repeating businesses.

Applying the principle of customer focus typically leads to researching and understanding customer needs and expectations, where the objectives of the organization then can be linked to. Those needs and expectations can be communicated throughout the organization, so that a customer relationship can be set up and managed systematically. Additionally, the customer satisfaction can be measured and actions can be planned based on the results. Finally, a balanced approach between satisfying customers and other interested parties (such as owners, employees, suppliers, financiers, local communities, or the society as a whole) would be ensured.

Principle 2: Leadership
Leaders establish the unity of purpose and direction of the organization. They should create and maintain the internal environment in which people can become fully involved in achieving the organization's objectives.

An important key benefit of the principle of leadership is the evaluation, alignment, and implementation of the activities in an organization in a unified way. People will understand the organization's goals and objectives and hence, be motivated towards them. Neither should the minimization of miscommunication between the levels of an organization be underestimated.

Applying the principle of leadership typically leads to a deeper insight into the needs of all interested parties including customers, owners, employees, suppliers, financiers, local communities, and the society as a whole. Through leadership a clear vision of the organization's future can be established and challenging goals and targets can be set. Additionally, it establishes trust and eliminates fear in the organization, which then creates and sustains shared values, fairness and ethical role models at all levels. People are provided with the required resources, training, and freedom to act with responsibility and accountability while their contributions are inspired, encouraged, and recognized.

Principle 3: Involvement of people
People at all levels are the essence of an organization and their full involvement enables their abilities to be used for the organization's benefit.

The key benefits of this principle are motivated, committed, and involved people within the organization, which leads to innovation and creativity in furthering the organization's objectives. People are accountable for their own performance and eager to participate in and contribute to continual improvement.

Applying the principle of involvement of people typically leads to people, who are understanding the importance of their contribution and role in the organization. Those people identify constraints to their performance and accept the ownership of problems and their responsibility for solving them. They evaluate their performance against their personal goals and objectives and actively seek opportunities to enhance their competence, knowledge, and experience. This enhancement typically results from knowledge and experience sharing as well as from open discussions about problems and issues.

Principle 4: Process approach
A desired result is achieved more efficiently when activities and related resources are managed as a process.

Key benefits of a process approach are lower costs and shorter cycle times through effective use of resources as well as improved, consistent, and predictable results. The organization's improvement opportunities can be focused and prioritized.

Applying the principle of process approach typically leads to a systematical definition of necessary activities to obtain a desired result. The responsibility and accountability for managing key activities can be established. Those key activities within and between the functions of the organization then can be identified, analyzed, and measured. If a focus lies on factors such as resources, methods, and materials, the key activities of the organization will be improved. Additionally, the risks, consequences, and impacts of activities on customers, suppliers, and other interested parties can be evaluated.

Principle 5: System approach to management
Identifying, understanding, and managing interrelated processes as a system contributes to the organization's effectiveness, and efficiency in achieving its objectives.

With this principle, key benefits include the integration and alignment of the processes that will best achieve the desired results and the ability to focus effort on those key processes. Additionally, it provides confidence to interested parties as to the consistency, effectiveness and efficiency of the organization.

Applying the principle of system approach to management typically leads to a structured system to achieve the organization's objectives in the most effective and efficient way. The interdependencies between the processes of the system are understood and structured approaches that harmonize and integrate processes are achieved. To achieve common objectives and thereby reducing cross-functional barriers, a better understanding of the roles and responsibilities is provided. The organizational capabilities are understood and hence, resource constraints prior to action are established. Specific activities within a system are targeted and defined and the system is continually improved through measurement and evaluation.

Principle 6: Continual improvement
Continual improvement of the organization's overall performance should be a permanent objective of the organization.

Key benefits of continual improvement are performance advantages through improved organizational capabilities as well as flexibility to react quickly to opportunities. The improvement activities at all levels are aligned to an organization's strategic intent.

Applying the principle of continual improvement typically leads to a consistent organization-wide approach to continual improvement of the organization's performance. People are provided with training in the methods and tools of continual improvement and continual improvement of products, processes, and systems is made an objective for every individual in the organization. Goals are established to guide and measure continual improvement, which is recognized and acknowledged.

Principle 7: Factual approach to decision making
Effective decisions are based on the analysis of data and information.

Key benefits are informed decisions and an increased ability to demonstrate the effectiveness of past decisions through references to factual records. Additionally, the ability to review, challenge, and change opinions and decisions is increased.

Applying the principle of factual approach to decision making typically leads to accurate and reliable data, which can be accessed by those who need it. This data is analyzed using valid methods and decision making and action-taking is based on factual analysis, balanced with experience and intuition.

Principle 8: Mutually beneficial supplier relationships
If an organization and its suppliers are interdependent a mutually beneficial relationship enhances the ability for both partners to create value.

A key benefit of this principle is an increased ability to create value for both parties and an optimization of costs and resources for the organization. Market changes or customer needs and expectations can be met in a flexible and fast way.

Applying the principle of mutually beneficial supplier relationships typically leads to established relationships that balance short-term gains with long-term considerations. Key suppliers can be identified and selected and expertise and resources can be pooled with partners. A clear and open communication makes information sharing possible and establishes future plans, joint development, and improvement activities. Finally, improvements and achievements by suppliers can be inspired, encouraged, and recognized.

There are many different ways of applying these quality management principles. How the principles are implemented in particular companies depends on the character of the organization and the specific challenges it faces. When successfully establishing the ISO 9000 requirements, a company can be certified by a third party certification centre. The certification is carried out by inspecting the company's quality management system processes in an audit.

2.2 CMMI (Capability Maturity Model Integration) [2]

The Capability Maturity Model Integration (CMMI) is a quality management model for systems and software engineering and successor of the well known CMM (Capability Maturity Model). The basis of CMMI is similar to the philosophy of the quality standard family ISO 9000. Hence, CMMI focuses especially on systems and software engineering, so it gives essential support in quality management for those areas. Additionally, CMMI supports continuous improvement, since it concerns a level based model.

Both, CMM and CMMI were developed at the Software Engineering Institute (SEI), Pittsburgh/USA, original on behalf of the American Department of Defense, whose aim was to evaluate the quality of software suppliers. The basis for ISO 9000 as well as for CMM and CMMI is the philosophy to improve results through the improvement of the work process. The main difference between ISO 9000 and CMM respectively CMMI is the focus on systems and software engineering and the level based model in CMMI, which follows the improvement through five levels from maturity level 1: Initialized to maturity level 5: Optimizing, while ISO 9000 only has the two levels fulfilled and not fulfilled.

Many companies already use CMM or CMMI to measure and improve their own processes or the processes of their software suppliers, among others BMW, Siemens, Bosch, E-Plus, EDS, and DB Systems. At the moment, many German car manufacturers debate on obliging CMM(I) or the similar SPICE for software engineering in the automobile sector.

2.2.1 The Five Maturity Levels of CMMI

In its level based representation, CMMI provides five maturity levels to identify the quality of a software engineering processes in a company. Figure 1 gives an overview of the maturity levels and their corresponding process characteristics.

Maturity Level 1: Initial

When a company's processes are identified as maturity level 1 the processes are carried out ad hoc or in a chaotic way. The processes are defined only little or not at all and the success of a company's project depends highly on the effort and competence of individual employees, so called heroics. In those projects relationships between disciplines are uncoordinated and the introduction of new technology is risky.

Maturity Level 2: Managed

Maturity level 2 implies that a company has introduced essential management processes which contain documented and stable estimating, planning, and commitment of a project's budget, time, and functionality. Problems in the project are recognized and corrected as they occur. The success of a company's project

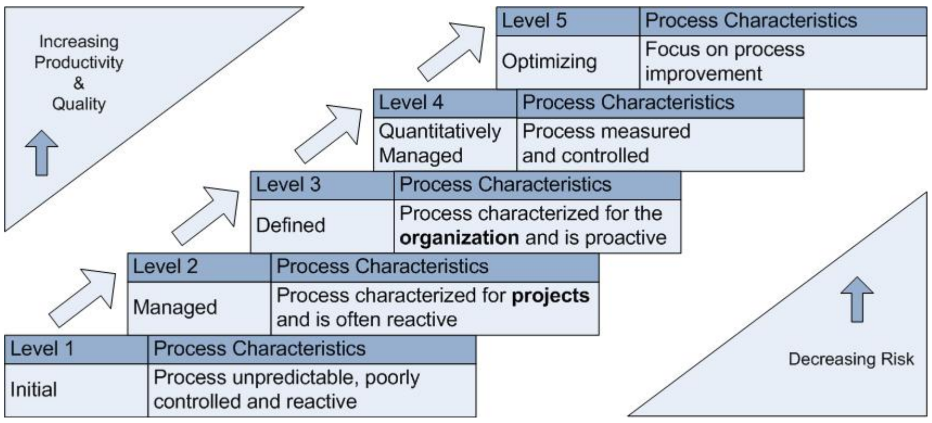

Increasing Productivity & Quality
Level 5
Process Characteristics
Optimizing
Focus on process improvement
Level 4
Process Characteristics
Quantitatively Managed
Process measured and controlled
Level 3
Process Characteristics
Defined
Process characterized for the **organization** and is proactive
Level 2
Process Characteristics
Managed
Process characterized for **projects** and is often reactive
Level 1
Process Characteristics
Initial
Process unpredictable, poorly controlled and reactive
Decreasing Risk

Fig. 1. The CMMI maturity levels

still depends on individual employees, but now a supporting management system is available and the employees are trained. In those projects technology supports are established, so that a stable software development is possible.

Maturity Level 3: Defined
In maturity level 3 the main emphasis shifts from individual projects to the entirely organization and its management activities. Most of the requirements in maturity level 3 aim at implementing a homogeneous process and engineering management for the whole organization, while maturity level 2 focuses largely on processes for individual projects. When an organization fulfills the requirements for maturity level 3, any problems in the project are anticipated and prevented or at least their impacts are minimized. Now the employees in the organization work together as a team and training for the team members is planned and provided according to roles. When new technologies appear, they are evaluated on a quantitative basis.

Maturity Level 4: Quantitatively managed
When an organization succeeded in implementing homogeneous processes, the next step is an intensive use of metrics and measurements to achieve a better decision basis for process improvement activities. In earlier maturity levels, metrics are already used, but only on a project base. In maturity level 4 an organization can extend the measurements on the entire organization to understand and stabilize its processes in a quantitative way. Within each project a strong sense of teamwork exists among the team members and sources of individual problems are understood and eliminated.

Maturity Level 5: Optimizing
Maturity level 5, the highest level in the CMMI model, focuses on a continu-

ous improvement. Every employee is involved in the process improvement and a strong sense of teamwork exists within the whole organization, not just within individual projects. New technologies are evaluated to determine their effect on quality and productivity and appropriate technologies are transferred into normal practice across the organization.

Except for level 1, each maturity level is decomposed into several key process areas, that indicate where an organization should focus to improve its software process. Table 1 shows the key process areas by maturity level. All key process areas for a level must be satisfied and the processes institutionalized to achieve a maturity level.

Table 1. The key process areas by maturity level

Maturity level	Key process area
5 Optimizing	Organizational Innovation and Deployment, Causal Analysis and Resolution
4 Quantitatively managed	Organizational Process Performance, Quantitative Project Management
3 Defined	Requirements Development, Technical Solution, Product Integration, Verification, Validation, Organizational Process Focus, Organizational Process Definition, Organizational Training, Integrated Project Management, Risk Management, Decision Analysis and Resolution
2 Managed	Requirements Management, Project Planning, Project Monitoring and Control, Supplier Agreement Management, Measurement and Analysis, Process and Product Quality Assurance, Configuration Management
1 Initial	

One exemplary goal in the *Process and Product Quality Assurance* key practice is the specific goal to *Objectively Evaluate Processes and Work Products.* Besides others, this goal includes the specific practice to *Objectively Evaluate Processes* [2]:

> **SG 1 Objectively Evaluate Processes and Work Products**
> Adherence of the performed process and associated work products and services to applicable process descriptions, standards, and procedures is objectively evaluated.
> **SP 1.1 Objectively Evaluate Processes**
> Objectively evaluate the designated performed processes against the applicable process descriptions, standards, and procedures.
> Objectivity in quality assurance evaluations is critical to the success of the project. A description of the quality assurance reporting chain and how it ensures objectivity should be defined.
> Typical Work Products

1. Evaluation reports
2. Noncompliance reports
3. Corrective actions

Subpractices

1. Promote an environment (created as part of project management) that encourages employee participation in identifying and reporting quality issues.
2. Establish and maintain clearly stated criteria for the evaluations. The intent of this subpractice is to provide criteria, based on business needs, such as the following:
   - What will be evaluated
   - When or how often a process will be evaluated
   - How the evaluation will be conducted
   - Who must be involved in the evaluation
3. Use the stated criteria to evaluate performed processes for adherence to process descriptions, standards, and procedures.
4. Identify each noncompliance found during the evaluation.
5. Identify lessons learned that could improve processes for future products and services.

CMMI is a well defined and structured method to measure the maturity level of a company's processes. By means of an appraisal it can be ascertained at which maturity level a company is situated and how the software processes can be further improved. The SEI has defined a special appraisal method for CMMI, SCAMPI (Standard CMMI Assessment Method for Process Improvement), which is the successor of the former method CBA-IPI (CMM-Based Appraisal for Internal Process Improvement) and has higher formal requirements [3]. A detailed introduction of the assessment method would go beyond the scope of this paper.

2.3 SPICE (Software Process Improvement and Capability dEtermination

Since 1993 ISO develops the process improvement model SPICE, which is published under ISO 15504. The intent of this model is to provide an extensive framework to assess and improve software processes by integrating and unifying approaches like ISO 9000 and CMMI. In some aspects SPICE follows the content, structure, and notation of CMMI.

The main focus of SPICE are process assessments, which provide a basis for both measuring the maturity level of processes as well as identifying modifications to improve a process. SPICE can be used to assess a company's own software engineering or to assess other companies, e. g. in the course of a supplier choice. However, the main focus is put on self assessment rather than on certification.

2.3.1 The Structure of SPICE

The reference model of SPICE considers two dimensions, the process dimension and the dimension of maturity levels. The process dimension is used to identify the completeness of processes, the dimension of maturity levels to determine their performance.

The process dimension

In the process dimension every process is assigned to one of five categories, which are distinguished in the following:

– The *Customer-Supplier process category* describes processes, which directly affect the customer. These are processes like software acquisition, customer service, or software delivery.
– The *Engineering process category* covers processes, which are used to define, design, implement, or maintain software products.
– The *Support process category* describes processes, which support other processes within a project, e. g. documentation, configuration management, or quality assurance.
– The *Management process category* includes processes, which are necessary to plan, control, or manage software projects. These are processes like project management, quality management, risk management, or supplier management.
– The *Organization process category* covers processes, which make it possible to define and achieve business objectives, e. g. by process definitions and improvements or human resource management.

Overall, 29 processes are defined and every process is assigned to one of those five process categories. Each process itself is described by base practices, which define the activities to achieve the particular process intention.

An exemplary base practice in the *Management process category* is the *Manage quality* process, which includes the following practices [4]:

> **PRO.5 Manage quality**
> The purpose of the *Manage quality* process is to manage the quality of the project's products and services to ensure the resulting products and services satisfy the customer.
> Managing quality involves identifying the required quality characteristics of the projects products, working to achieve this quality, and demonstrating that this quality was achieved.
> Inputs are the customer requirements and selected elements of the software project plans (see process PRO.2). Outputs should be integrated into the software project plans.
> Note: There is another process with a similar name, "Perform quality assurance" SUP.3. Process PRO.5 focuses on identifying what needs to be done to build quality into the products and establishing management controls to ensure this gets done; whereas SUP.3 focuses more on an audit and review approach and on ensuring compliance.

PRO.5.1 Establish quality goals. Based on the customer's requirements for quality, establish quality goals for various checkpoints within the project's software life cycle (e.g. at the end of each phase).
PRO.5.2 Define quality metrics. Define metrics that measure the results of project activities to help assess whether the relevant quality goals have been achieved.
PRO.5.3 Identify quality activities. For each quality goal, identify activities which will help achieve that quality goal and integrate these activities into the software life cycle model.
PRO.5.4 Perform quality activities. Perform the identified quality activities.
PRO.5.5 Assess quality. At the identified checkpoints within the project's software life cycle, apply the defined quality metrics to assess whether the relevant quality goals have been achieved.
PRO.5.6 Take corrective action. When quality goals are not achieved, take corrective action.
Note: The corrective action can involve fixing the product generated by a particular project activity or changing the planned set of activities in order to better achieve the quality goals or both.

The dimension of maturity levels
SPICE distinguishes, similar to the five maturity levels in CMMI, six maturity levels, which can be used to assess the completeness and performance of processes in a company. In addition to the five maturity levels in CMMI, SPICE has a supplemental maturity level 1, which is especially meaningful for smaller organizations [5]. In each case, the next higher maturity level indicates suggestions for the process improvement.

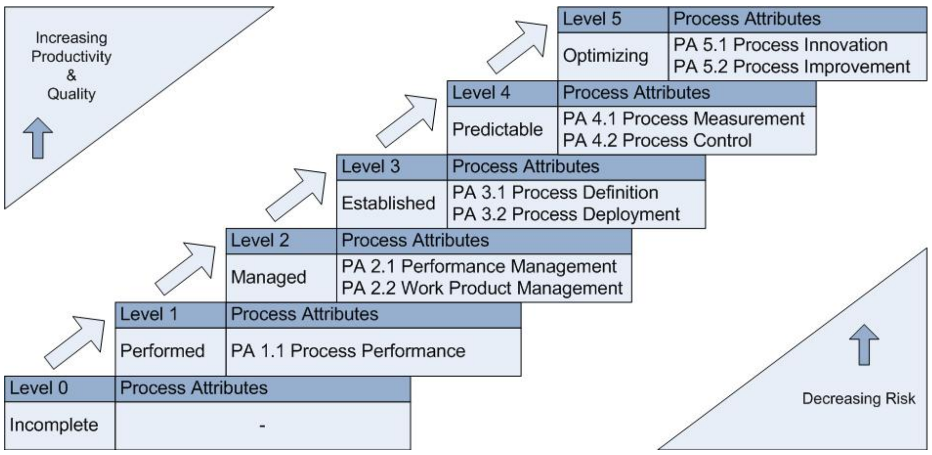


Fig. 2. The SPICE maturity levels

The performance of the 29 processes described in the process dimension is assessed by the nine process attributes in the dimension of maturity levels. Every process attribute represents a measurable characteristic of the process and is rated in a four-stage scale:

N	Not achieved
P	Partially achieved
L	Largely achieved
F	Fully achieved

Every process attribute is assigned to one maturity level and the process maturity level is calculated from the process attribute assessments. To achieve a maturity level, all process attributes of the concerning maturity level have to be at least largely achieved, and all process attributes of the subjacent maturity levels have to be fully achieved. Figure 2 gives an overview of the process attributes assigned to the respective maturity levels.

2.4 Comparison of the Introduced Models

ISO 9000, CMMI, and SPICE all have the same fundamental idea and the requirements of CMMI can be mapped to those of ISO 9000 (a table is available at the SEI [6]). However, in contrast to ISO 9000, CMMI and SPICE are especially designed for the software engineering process. ISO 9000 covers the organization as a whole, while CMMI and SPICE restrict to concrete process areas and practices. Additionally, an assessment according to ISO 9000 does not lead to some maturity level of the kind of CMMI or SPICE.

Both methods, CMMI (with the newer assessment method SCAMPI) and SPICE, use a standardized list of questions. However, in a SPICE questionnaire four possible answers for an assessment question are possible, not achieved, partially achieved, largely achieved, and fully achieved, while at CMMI there exists no differentiation, the result is determined only by yes/no answers. Since SPICE is an ISO norm it is more common in Europe than in the USA. SPICE forms, for instance, a de facto standard for the assessment of suppliers in the German automobile sector.

CMMI includes actual directives how to improve a process, while SPICE does not carry along any directives and hence, does not force a specific order for the process improvements. Rather, the catalog of measures is derived from the experiences of the accompanied consultancy according to the outcomes of the assessments. Typically, a company begins the improvement in those areas which show major deficits.

As already mentioned the two methods also differ in the complexity of the assessments: CMMI requires that at least one employer of the company is familiar with the method. On the other hand, SPICE does not demand an adequate education. Resultant, CMMI has a deeper view at the analysis, the pending improvements, and the organizational aspects. Therefore, a company should figure out if this depth is necessary when deciding about a suitable assessment method or if the resulting effort is not adequate for the benefit. For most small to medium-sized companies, SPICE is sufficient to gain selective process improvements.

3 How to Measure Process Quality [7]

Since CMMI has been developed by the Software Engineering Institute (SEI) in 1984, it has often been stated that companies which use and improve defined software development processes can produce software of a higher quality.

Unfortunately, there have not yet been made many examinations if those assertions are true or how the increased software quality could be measured. To fill this lack with information Caspers Jones has examined project data between 1984 and 2002 together with colleagues at Software Productivity Research (SPR) [7]. Using questionnaires Jones and his colleagues have appraised data from more than 10,000 software projects at all five levels of the SEI capability maturity model (CMM) for gathering both qualitative and quantitative information. Additionally, many projects were examined which do not use the SEI CMM.

Most examined companies followed a six-stage process improvement program, which is typically initiated by a formal process assessment and a baseline:

Stage 0: Software Process Assessment, Baseline, and Benchmark
Stage 1: Focus on Management Technologies
Stage 2: Focus on Software Processes and Methodologies
Stage 3: Focus on New Tools and Approaches
Stage 4: Focus on Infrastructure and Specialization
Stage 5: Focus on Reusability
Stage 6: Focus on Industry Leadership

These six stages can serve as a structure to guide through a process improvement process. Note that the assessment itself is not a real part of the improvement, so it is outside the six numbered improvement stages. An important aspect and not to forget is that every company or project is different and hence, the six stages have to be adapted to the particular working environment.

In the following this approach will serve as an example how process quality improvement can be measured using the function points metric. We will focus on the assessment, baseline, and benchmark, since a discussion of the six stages would go beyond the scope of this study and would be similar to the introduction of the maturity levels in CMMI or SPICE.

3.1 Stage 0: Software Process Assessment, Baseline, and Benchmark

It is important to point out once more that this stage is outside the six process improvement phases. Neither an assessment nor a baseline itself do improve a tithe of the process, but anyhow, many companies forget this point and tend to stop doing anything after an initial assessment is carried out and a baseline is determined.

Every process improvement should begin with a formal assessment of the current process and the establishment of a quantitative baseline of current productivity and quality levels. The assessment helps to identify the strengths and weaknesses of the process associated with software and is often carried out by consulting groups. The baseline provides a basis for productivity, schedules, costs, quality, and user satisfaction in order to benchmark the future improvement progress. As described in this chapter, function points are often used for baseline data collection, since they cover a wide range of activities. Besides comparing collected data from the baseline with data from a future point, it can also be considered to compare a company with another company in the same business sector. Third-party consulting groups often have large collections of software data from many companies and industries.

3.2 A Short Discussion of Function Point Metrics

Since this chapter will deal with process quality measurements using function point metrics, a short discussion of function point metrics may be useful. Function point metrics were developed by A. J. Albrecht and colleagues at IBM in the mid 1970s. A function point is defined as an end-user business function, such as a query for an input. Basic function points are categorized into five groups of external attributes: inputs, outputs, inquiries, logical files, and interfaces. Since the counting rules for function points are complex, precise counting of function points is normally carried out by specialists who have passed a certain certification, conferred by the International Function Point User Group organization (IFPUG).

Compared to the well-known software metric lines of code (LOC), function points offer some significant advantages for baselines and benchmarks. An important difference to state is that function points can be achieved before the coding starts, while LOC metrics can only be used once the project is ready and have not been useful for measuring the volume of specifications, the contributions of project management, or the defects found in requirements and design documents. Further more, LOC do not represent the same in different programming languages. Indeed, for some programming languages such as Visual Basic, there are no effective LOC counting rules. Subsidiary, there exist conversion tables, which allow to convert function points to LOC. However, since there is no reliable relation between function points and lines of code, every conversion produces a certain fuzziness. Since function point metrics are so powerful and can measure all software activities from the requirements engineering over the coding to testing and management they have become a de facto standard for software baselines and benchmarks.

3.3 Measuring Process Improvements at Activity Levels

In this section we will compare the baseline of a hypothetical project of 1,000 function points (roughly 125,000 C source statements) at SEI CMM maturity level 1 and level 3. The software application of the project is a systems software, written in C. The work hours per month in the project add up to 132 and the burdened average monthly salary is $7,500.

Activity	Work Hours per FP	Staff	Effort (Person Months)	Schedule (Calendar Months)	Costs by Activity	Percent of Costs
Requirements	1.20	2.00	9.09	4.55	$68,182	4%
Design	2.93	3.33	22.22	6.67	$166,667	11%
Design reviews	0.38	4.00	2.86	0.71	$21,429	1%
Coding	7.76	6.67	58.82	8.82	$441,176	29%
Code inspections	0.53	8.00	4.00	0.50	$30,000	2%
Testing	8.25	6.67	62.50	9.38	$468,750	31%
Quality Assurance	1.32	1.00	10.00	10.00	$75,000	5%
Documentation	1.10	1.00	8.33	8.33	$62,500	4%
Management	3.57	1.00	27.03	27.03	$202,703	13%
Totals	27.04	6.33	204.85	32.35	$1,536,406	100%
FP per month	4.88					
LOC per month	610					
Cost per FP	$1,536.41					
Cost per LOC	$12.29					

Table 2. Example of Activity-Based Cost Analysis for SEI CMM Level 1

Table 2 illustrates the project for a typical company at CMM Level 1. As already stated in chapter 1 level 1 organizations are not very sophisticated in software development techniques and thus, often have missed schedules, cost overruns, and software products of poor quality. Noticeable is the characteristic that level 1 organizations spend most time and budget in testing the software system, because they usually have excessive defect levels and do not carry out defect prevention or pretest reviews and inspections.

In contrast, table 3 illustrates the same software project for a level 3 company. As already illustrated in table 1, the development from CMMI maturity level 1 to level 3 results in implementing techniques like verification and validation. Therefore, testing has eased significantly in both time and budget, because those methods for defect prevention, pretest design reviews, and code inspections are established now.

Although those comparisons seem to be universally valid statements, it has to be pointed out, that it is difficult to gain objective and meaningful statistics, especially because the pictured projects are only fictive ones. Thus, every company should establish its own data using its own assessment and baseline studies instead of counting too much on reports like the presented.

Activity	Work Hours per FP	Staff	Effort (Person Months)	Schedule (Calendar Months)	Costs by Activity	Percent of Costs
Requirements	1.06	2.00	8.00	4.00	$60,000	5%
Design	2.64	3.33	20.00	6.00	$150,000	12%
Design reviews	0.88	4.00	6.67	1.67	$50,000	4%
Coding	6.00	6.67	45.45	6.82	$340,909	28%
Code inspections	1.06	8.00	8.00	1.00	$60,000	5%
Testing	3.30	6.67	25.00	3.75	$187,500	15%
Quality Assurance	2.20	1.00	16.67	16.67	$125,000	10%
Documentation	1.10	1.00	8.33	8.33	$62,500	5%
Management	3.30	1.00	25.00	25.00	$187,500	15%
Totals	21.53	6.33	163.12	25.76	$1,223,409	100%
FP per month	6.13					
LOC per month	766					
Cost per FP	$1,223.41					
Cost per LOC	$9.79					

Table 3. Example of Activity-Based Cost Analysis for SEI CMM Level 3

Table 4. Side-by-side comparison of activity-based costs

Activity	SEI CMM Level 1	SEI CMM Level 3	Variance in Costs	Variance Percent
Requirements	$68,182	$60,000	-$8,182	-12.00%
Design	$166,667	$150,000	-$16,667	-10.00%
Design reviews	$21,429	$50,000	$28,571	133.33%
Coding	$441,176	$340,909	-$100,267	-22.73%
Code inspections	$30,000	$60,000	$30,000	100%
Testing	$468,750	$187,500	-$281,250	-60.00%
Quality Assurance	$75,000	$125,000	$50,000	66.67%
Documentation	$62,500	$62,500	$0	0.00%
Management	$202,703	$187,500	-$15,203	-7.50%
Totals	$1,536,406	$1,223,409	-$312,997	-20.37%
Cost per FP	$1,536.41	$1,223.41	-$313.00	-20.73%
Cost per LOC	$12.29	$9.97	-$2.50	-20.73%

A look on table 4, which illustrates the side-by-side analysis of the costs, shows that an overall cost reduction of 20% has been achieved by the process improvement from level 1 to level 3. As anticipated, most of the savings occur during the testing phase. However, the costs for code inspections are higher at level 3 than at level 1 and some costs, like those for user documentation, kept the same in both scenarios.

Level	Potential Defects	Removal Efficiency	Delivered Defects	Defects per Function Point	Defects per KLOC
SEL Level 1	6150	85.01%	922	0.92	7.38
SEI Level 3	3500	95.34%	163	0.16	1.30

Table 5. SEI CMM Level 1 and Level 3 Defect Differences

Besides those project budget differences Jones and his colleagues also took a look at the quality of the resulting software products. The improvement of both defect prevention and defect removal approaches in level 3 companies guides to a significant reduction in delivered defects and hence, to the ability to use shorter and more cost-effective development cycles, which factor defect prevention into the software development. To gain an overview of the differences in defect potentials, defect removal efficiency levels, and delivered defects, table 5 compares the defect difference of the level 1 company with the level 3 company. Here, potential defects means effects which are likely to be encountered from the start of requirements analysis through at least one year of customer use.

4 Discussion of the Quality Management Models

A process assessment with a resulting maturity level systematically indicates strengths and weaknesses in a company's processes and identifies possibilities for improvements. Additionally, it is important to have a general framework for the assessment of software processes so that a company has a chance to compare itself with other companies in the same business area. However, process improvement models do not only have strengths but also weaknesses. Hence, this chapter deals with problems with CMMI and SPICE.

One of the main objections to process improvement models is that many models, e. g. CMMI, solely concentrate on the process as the main factor in software development, leaving out people and technology. Thus, instituting CMMI, whether in maturity level 1 or 5, is no guarantee that a software process will succeed or, more bluntly, that the quality of the developed software will increase, if the process is promoted over all other issues. Especially coding software, one main activity in software engineering, is not considered in a really extensive way in all three presented methods. Furthermore, a too strict focus on the process holds the risk to work more on the process than on the software itself.

One point, regarded to all existing process improvement models, is the fact that an organization can tend to "slip back" to their previous methods of doing business after achieving their rating. So an organization can solely implement the model to achieve the aspired maturity level, e. g. if it wants to win a federal government contract, which requires a specific maturity level. Needless to say, that those organizations sell themselves short by paying money and effort to achieve the level but not taking the full advantage of the improvement. To gather long-term process improvement the organization's culture must change to a more open-minded in order to be able to decide whether it is useful to change processes or not.

One last issue is the fact that the amount of process improvement models is increasing more and more. Besides the three well known models presented in this article there exist quite a number of other, more or less similar models, e. g. Bootstrap, SPIRE, TickIT, or Trillium. Hence, choosing the most suitable approach for a company's software process is difficult.

5 Summary

The paper on hand gave a short introduction to the process quality standard ISO 9000 and the two process improvement models CMMI and SPICE. Besides, a method to measure process improvement using the function points metric was introduced. A subsequent discussion of process improvement models showed that especially the two younger improvement models CMMI and SPICE lack some important aspects of process improvement. Thus, neither in ISO 9000 nor in CMMI or SPICE the improvement considers coding, which is however a core activity in software engineering.

Nevertheless, implementing CMMI or SPICE can significantly raise the probability of success in a software process. Every company has to find a suitable model to meet its own demands and the effort put in the process improvement has to be adequate to the project size and complexity.

References

1. ISO 9000 and ISO 14000 Website `http://www.iso.org/iso/iso_catalogue/management_standards/iso_9000_iso_14000.htm`; visited in April 2008.
2. CMMI Product Team, S.E.I.: Cmmi for development. Technical Report 1.2, Carnegie Mellon University, Software Engineering Institute (2006)
3. Kneuper, R.: CMMI. Verbesserung von Softwareprozessen mit Capibility Maturity Model Integration, 2. Auflage. dpunkt.verlag (2006)
4. SPICE Project: Software process assessment - part 2 : A model for process management. Technical Report 1.00, SPICE Project (1995)
5. Balzert, H.: Lehrbuch der Software Technik. Spektrum Akademischer Verlag (1998)
6. Mutafelija, B., Stromberg, H.: Iso 9001:2000 - cmmi v1.1 mappings. Technical report, Carnegie Mellon Software Engineering Institute (2003)
7. Kan, S.H.: Metrics and Models in Software Quality Engineering, Second Edition. Addison Wesley (2002)

8. Carnegie Mellon University, S.E.I.: The Capability Maturity Model: Guidelines for Improving the Software Process. Addison Wesley (1995)
9. Bollinger, T.B., McGowan, C.: A critical look at software capability evaluations. IEEE Software **8**(4) (1991)

Sustainable Change in Organizations

Stefan Puchner

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
puchner@in.tum.de

Abstract. Software quality is not just a matter of introducing tools and frameworks. It is about changing the behavior of developers in order to produce software of better quality. A tool cannot just "switch on" software quality; all it can do is to support developers in the behavior demanded from them. Change management is necessary to really "change" behavior.

1 Introduction

In a lot of businesses today, software makes up for a considerable amount of costs. That is not only due to development or deployment of new software, but also due to operation and maintenance of existing systems. That is especially true for very large systems, which cause an enormous investment by the time introduced and can therefore not be replaced very easily. They are running up to several decades but the requirements and the environment are constantly changing. As a result, the system needs to be changed constantly during its runtime (often referred to as maintenance). That usually causes much higher investments than the initial development and deployment of the system. If the software is of low quality, it is even harder to maintain and will consequently increase the cost more than necessary. The same is true for other cost drivers, like operation. A non-scalable algorithm in a crucial IT-system of a growing company might not show at the time of the introduction of the system. However, after some time, when the firm has grown bigger and the IT-system has to handle multiple times as many operations, this algorithm might become a costly bottleneck for the company's operations.

However, many companies are not successfully dealing with quality issues yet. This might be due to missing methodology knowledge regarding software quality or due to the challenges an organization has to overcome when implementing such methodologies. Implementing new methodologies means to alter established approaches and work habits. Because changing employees' work habits does not seem to be a big challenge, many companies underestimate this issue and consequently fail to change. In management literature the process of altering organizations is called change management. It is not the same as what is referred to as change management in a product development context. Latter describes how changes in a product or product concept are managed, whereas change management as discussed in this paper is about changing an organization itself. Popular examples for such changes are process reengineering, restructuring, cultural changes, as well as quality management. [1]

Software quality is not just a matter of introducing tools and frameworks. It is about changing the behavior of developers in order to produce software of better quality. A tool cannot just "switch on" software quality; all it can do is to support developers in the behavior they should adapt to. So when software quality should be improved, the challenge is not so much which tools to choose. More important are challenges like making developers aware of the necessity to change their behavior, or empowering developers to be able to live the new behavior and processes demanded from them.

There are many cases in literature where companies failed in changing themselves, although the reason for the change and the change itself were not generally viewed as controversial. An example regarding software quality can easily be constructed:

Let's consider a company introducing the tool Checkstyle at the repository level to improve software quality (`http://checkstyle.sourceforge.net`). A week before Checkstyle is activated, the administrator of the repository servers sends out an e-mail saying that developers will get daily automatically generated messages as long as the code they checked in does not match common coding conventions. What will be the results of such an approach? Developers will start to feel insecure, because they might not know what this change will mean for them, for their daily behavior. Questions like the following arise: What are those coding standards exactly? Will the group leader be serious about violations of those standards? Will it take more time to do the same work when following those standards? Which consequences will I suffer when violating those standards? Furthermore, the employees might not really know whom to talk to and ask questions regarding this change. The group leader was not informed except through the very same e-mail and does not know more than anyone else. The administrator sending the mail does not know the implications for the developers either, he was just installing the tool and sending the mail because he was obliged to. Last but not least, the way the company did software development for many years worked just fine for the employees. Consequently, they don't see the necessity for coding standards and therefore the rational of introducing Checkstyle. Developers will get angry about management for all the trouble they caused without a reason.

Research about this phenomenon of important changes failing in realization was conducted as early as 1947, when [2] developed a very basic model for organizational change. Lewin suggest a 3-phase model of 'unfreezing', 'moving', and 'freezing' group standards. Newer literature like [3], [4], and [5] suggests more sophisticated approaches on how to deal with change in nowadays companies, influenced by increasing competition.

2 Suggested Approach

This report is mainly based on findings of general change management literature. After summarizing the relevant findings in each section, they are applied to an imaginary example case about introducing software quality into a company.

The steps this paper suggests in order for organizations to introduce software quality efforts are the following:

– Convince Stakeholders by Establishing a Sense of Urgency
– Create a Guiding Coalition
– Communicate Change
– Establish Short-term Wins
– Make Change Permanent

This paper will refer to different groups in companies. Going down the hierarchy those groups will be called: upper management (CEO, CTO, board of directors, etc.), middle management, group leaders, and developers (programmers, architects, etc.).

3 Convince Stakeholders by Establishing a Sense of Urgency

Going through a change means additional effort for everyone in the organization. If there is no sense of urgency among employees, most of them will not be willing to make that effort, because they don't see a rational for the change. They don't want to change a - in their mind - running system, but rather keep their behavior 'frozen'.

Consequently, the first step when introducing change in an organization should always be getting all stakeholders to buy-in the urgency of the situation. [2] refers to this phase as 'unfreezing' group standards. According to [1] over 50% of companies fail in this phase. Even if change leaders are aware of the fact that a sense of urgency has to be established actively, they might "think that the people are, for the most part, pretty smart, so all you have to do is give them the facts about poor product quality, sliding financial results, or lack of productivity growth." [4] However, [5] states that facts alone will in most cases not cause enough sense of urgency. Not only the mind, but also the feelings of employees have to be addressed in order to effectively raise urgency. Many firms fail in this phase, because change leaders "underestimate how hard it can be to drive people out of their comfort zones." [1]

3.1 Actions Driving Urgency

[4] suggests rather drastic actions to raise the urgency level. In the field of introducing software quality, this would be setting time goals for software modification to a level, which makes it impossible to achieve them without improving

software quality seriously. Another rather drastic action to increase the sense of urgency could be comparing the software quality metrics generated by different departments.

In contrast to this approach, [5] describes actions, which rather touch the emotional side of people, in order to make them aware of the necessity of change. He emphasizes on his finding that "people change what they do[,] less because they are given analysis that shifts their thinking than because they are shown a truth that influences their feelings." According to [5] "making a business case [is] not inherently bad. But when you find [data and thinking] at the beginning of successful large-scale change, they are aimed at supporting a more powerful method - one based on helping people to see a truth, feel differently, and then act with more urgency." Therefore, [5] suggest addressing feelings by giving concrete, dramatic, and surprising information and examples. Depending on the situation, giving a point of view, which is different from the one of the audience has, can help. This could be customer opinions for instance. On the same line, [6] encourages convincing stakeholders about software quality projects by giving examples like code snippets, in between facts.

3.2 Different stakeholders have different pains

A change leader has to be aware that different stakeholders have different pains and may therefore have to be addressed differently in order to create a sense of urgency to them. To address every stakeholder's specific, sensible areas, the first thing to do is to identify the stakeholders. This is followed by an analysis of which aspects of the change is most crucial for every individual stakeholder. Based on this information, activities of how to raise urgency in the individual stakeholders can be derived.

3.3 Example

Let's consider a software quality project that wants to reengineer the company's main product, an ERP system. A competitor is about to gain more and more market share, because its ERP software is outperforming the company's product in various aspects. The goal of the project would be more efficient usage of hardware resources, higher reliability, and faster handling of change requests.

To establish urgency in a "data and thinking" kind of way, the methodologies presented in [6] can be used. That includes measuring the software quality of the ERP system in different ways, estimating what reengineering the product will cost, and estimating the continuous cost savings the improved quality will generate. This will create a solid business case for improving software quality of the ERP system.

Approaches to raise urgency, which rely more on the feelings of employees and that are individually aimed on stakeholders, could be the following.

– For upper management a negative vision of what is likely to happen to the company, if it continues without increased software quality, might be

convincing. An example of such a negative vision is, a competitor, who is outperforming the company's product in every aspect (efficiency, reliability, fast changes), which will lead to a rapid drop in the company's market share.
- The following approach can be used if developers feel the urgency of software quality, but management or group leaders do not. Management would urge developers to deliver changes and features in the least possible time, instead of focusing on software quality. To convince management, a video of a developer doing changes on code could be shown. Because of multiple cloning of code the developer has to do the same change several times. This means tedious and error prone full-text-searches. Such a scenario will hopefully shock management, and therefore increase the sense of urgency about the software quality issue.
- To raise urgency in developers, a video showing a worried customer could be shown to them. In the video the customer complains about too many bugs and the slow change process. In addition, the customer describes what went wrong in the customer's operative business, just because of the ERP system not working properly. [5] points out, that employees often think they do everything right and know everything better than the customer. Such an ignorant attitude comes from years of success of a product. In this situation, facts and data are likely to be ignored. The approach of not presenting plain data but touching emotions is according to [5] more effective in changing the sense of urgency in an employee.

How far a sense of urgency should be established in users and customers depends on how much they are directly affected by the change process itself. In many cases it might not be the best way to make customers aware of hard times a company is going to face during the change effort. Using customers to increase the sense of urgency in the own organization, however, can have enormous positive impact as well.

3.4 Comments on Consulting

If acting as an outside consultant to the company, which is undergoing a change, the initial step before convincing stakeholders is to sell the consulting project, of course. Before a sense of urgency is to be established on a broad basis, the paying client in the firm needs to be convinced of the necessity of higher software quality in his organization, first. This will usually be one of the stakeholders named above and the activities to convince this stakeholder are the same, no matter if before or after landing the project.

Regarding consulting in change efforts, [7] drives attention to being aware of clients delegating the change in a company on consultants too much (in the context of business process reengineering). For a successful change, management needs to transform its own management styles or priorities as well. A completely 'outsourced' change management will hardly be successful.

4 Create a Guiding Coalition

Once the sense of urgency for the change project is apparent to the organization, a team that is actually leading the change needs to be formed [4]. Common mistakes are making the CEO the one and only responsible person, having a too small team, or a team with too little power. Many change projects are too complex in order to be handled by a very small team or even one person. Furthermore, changes in organizations often take years. If the members of the guiding team are not powerful enough, this will result in a rather bad standing in times of high resistance against the change.

4.1 Characters on the team

According to [4] the people with the following characteristics need to be a part of the guiding team:

- Position power: In order to make the team capable of acting during all times, rather powerful people should be part of it.
- Broad expertise: All expertise needed during the project should be available in the team, to enable understanding of issues and right decisions in a timely manner.
- Credibility: It is crucial to have people on the team, which can effectively communicate the decisions and plans of the team to employees.
- Management: Management skills are needed to make plans on how to progress and to keep track of the progress in the end.
- Leadership: While managers are controlling the change process, leaders are driving it. They are important for creating visions and giving direction.

When creating the guiding coalition it is also important to think of who will potentially be resisting the change if not in the team, and who will need to be on the team in order to reach crucial groups.

4.2 Teamwork is crucial

For the guiding coalition to work effectively, they have to work together smoothly and appear as a unit to the outside. Typical no-goes for teamwork are people not really supporting the change, managers with big egos, or people creating mistrust in the team. Teambuilding events help to familiarize the team members with each other's working and communication style. Even more important, these events help to establish a common goal. The role of the common goal is extremely important. Every team member has to have a common understanding of where the change is supposed to go and how the change should take place. If that is not the case the team members will either work in different directions or will have to consult each other much too often, resulting in low efficiency.

4.3 Example

Referring to the reengineering example used before, the following thoughts should be kept in mind when a guiding coalition is built.

- Expertise in software quality as well as in the ERP market will be needed in order to make the right decisions in a timely manner. But also group leaders who know most about the actual work environment and processes in the company are important expertise holders. If there is not enough expertise in software quality reengineering, it might also be bought externally, in form of hired consultants.
- Higher and middle management is often not very credible to most employees; group leaders might be the better choice for credible communication. Developers might rather listen to group leaders when talking about new conventions on how to name variables than to the CFO.

Once a guiding coalition is established and it has figured out the actions of how to go through the change, the next challenge is to communicate the change actions successfully.

5 Communicate Change

Achieving software quality means changing the way software is developed and therefore it means changing how developers do their job. Changing the way, in which developers act, is a matter of communication. The third step, after establishing a sense of urgency and creating the guiding coalition, is to communicate what employees should actually do differently.

5.1 Don't rely on middle management

A very common mistake, which [3] emphasizes on, is to rely on middle management to implement change. The individuals in management have different interest. Some of them might be more interested in "quick and dirty" solutions, because they want to finish their project on time rather than having a high quality. High quality might not pay of before the software is modified again, when the manager has already moved on.

A second reason [3] names, is that there is no real incentive for middle management to communicate the change. Rather than giving precise instructions what to change and initiating qualification programs, middle management will just tell group leaders and developers to produce quality. That way, middle management could push the responsibility down, once low quality shows, because it was actually the developer that did something wrong, not middle management. In fact this is even worse than not communicating the quality requirements at all. This behavior will create insecurity and cast rumors about what developers are supposed to change, and what will happen if they don't change.

[4]argues that middle management usually lacks leadership qualification and culture and is rather experienced in handling bureaucracy. According to him change is 70 to 90 percent leadership and only 10 to 30 percent middle management. However, most managers are only able to manage and not to lead .He even says middle management is usually focused and arrogant, because of the past success they experienced in the company. This will cause them to resist change out of false self-confidence.

5.2 Communicate to group leader directly

Observations of various sources teach that resistance to change is often a result of subjective evaluation of change [8]. Therefore, it is important that developers get the message of the change right. Specific and precise work instructions will lead to a change, blurred and unspecific information dripping through middle management will not. Furthermore, direct communication to group leaders will allow incorporating feedback directly. Group leaders know the work environment and the developers best; therefore they can give the most valuable feedback the plans of the guiding coalition.

So why not communicate to developers directly in the first place? [3] justifies communicating to group leader instead of frontline employees with limited motivation and capabilities to receive instructions from up high. A group leader, however, knows how to communicate to individuals in his 'troop' much better. At that point, we should keep in mind, when [3] writes about frontline employees, he thinks of very low skilled work force. For example he writes about how workers would not find France on a map or that a majority of Americans has not read a single book after graduating from high school (chapter 17, page 160). So when we talk about software developers, who are likely to be higher educated employees than [3] assumes, we really should consider communicating to employees directly as well.

However, [3] gives two reasons why not communicating to frontline employees directly makes sense anyway. First, a reason for big companies is scalability. If the guiding coalition would have to communicate with 50,000 individuals, they will be very busy with just doing that. Having a layer of multipliers in-between to cascade the communication, will reduce the effort enormously. But does it have to be group leaders then? [3] recommends communicating to employees through the most trusted source, which is usually group leaders. Consultants come from outside; the heterogeneity of corporate cultures might not help for developers accepting instructions. If the organization has a more complicated corporate structure than assumed here (e.g. project oriented), network analysis could be a helpful way to identify opinion leaders, who might be a good disseminator for the change.

Second, communicating through group leaders will empower them and therefore help them establish change. In the end, the group leader is the contact person and the controlling person for the developers. If employees get important information from their group leader, rather than messages from management or a company newsletter, this will foster the group leader's authority. Eventually, authority of a controlling person will empower this person to establish change.

[4] writes in this context of empowering employees and making their interests the ones of the company, rather than the ones of their superior middle-level managers, who might rather resist the change. The point, however, is not to give power, but rather to remove obstacles [5]. With empowering employees, to get them the information they need and build their self-confidence, employees will be enabled to support the change in their area with whatever behavior is needed to do so.

Talking to supervisors does not mean to ignore management. Superiors, who seriously disempower their subordinates, should be "retooled" by giving them new jobs that clearly show the need for change [5].

5.3 Communicating a Vision

While [3] suggests not communicating anything, but precise plans and instructions on how to make the change, [4] wants the guiding coalition to create a vision and to communicate this vision first.

[3], on the one hand, argues against such texts like vision statements. Vision statements are by definition not very specific. According to [3], that causes employees to get confused and have an insecure feeling about what will be expected from them and what will happen to them as a consequence. Feeling insecure will rather make employees resistant to change than supportive.

[4], on the other hand, describes the positive impact a vision can have as a guideline for the whole company. A vision can be the rational basis for decisions and a strong argument against prevailing practices. Furthermore, it might enable much more efficient coordination. Ideally, with a common vision, everybody is heading in the same direction without having to have endless discussions. Here again, the vision needs to serve the interests of all stakeholders. In order to not confuse employees with the vision, but rather showing them a direction, [5] suggest preparing an extensive Q&A collection as backup for the presenters of the vision. So questions can be answered precisely, shortly, and convincing. That helps reducing insecurity and creating buy-in.

However, [4] also argues that an "ineffective vision is worse than no vision". When deciding to go with a vision statement and communicate it, one should know exactly that he got the vision right.

5.4 Example

Let's assume a company wants to adopt the tool Checkstyle to improve software quality. In order to communicate the change to group leaders or developers directly, seminars should be administered. That way, developers get first hand information about how their environment will change (e.g. on the repository level) and how they are supposed to change their behavior. At the same time, the conductors of the seminar will be able to collect direct feedback from developers.

Furthermore, the planned support for developers should be discussed. That could be technical mechanisms, like an IDE plug-in that immediately warns about style violations, but it could also be an internal hotline to answer questions about the changes.

6 Establish Short-term Wins

As [9] suggest, change is usually disruptive to a business, because it takes some time until new habits are fully learned and adopted. Even if the change was communicated properly, a company will be performing lower for a while after implementing the change. That means a lot of effort on the one hand, but no or worse outcome until the learning curve reaches a certain level on the other hand. Such a situation is likely to foster resistance against the change. To weaken resistance, it is a good strategy to plan for short-term wins.

The roles of short-terms wins in a change project are according to [4]:

- Success evidence to justify the effort and to show that the transformation is going somewhere
- Rewards for employees working hard on the change
- Feedback to fine-tune change strategies
- Arguments against change resisters
- Opportunities to build momentum a make more employees support the change
- Means of pressure to keep sense of urgency

The last point is especially important. After the momentum of the first urging actions and announcements about the change fade, organizations tend to fall in the 'frozen' mode again. To not let this happen, making employees responsible for reaching short-term goals will make a difference. Furthermore, it will also ensure that the steps necessary for the change are executed timely and the transformation progresses.

It is essential to really 'plan' on short-term wins, instead of just hoping that wins will be visible at some point [4]. First, the stakeholders who should be convinced of the change with the help of a short-term win are to be identified. Second, the short-term win should be planned so that it will address those stakeholders' issues.

In order for short-term wins to be effective there are three criteria, one should plan short-term wins on. First, short-term wins need to be visible for many people and not just executives. Second, short-term wins need to be unambiguous, so that they cannot be turned down by any arguments. Third, short-term wins need to be clearly related to the change effort [4].

6.1 Example

Let's consider our reengineering example again. Improving the software quality of the ERP system will include, among others, refactoring identifiers, changing the architecture, and merging redundant code. These changes will, in the beginning, lower the performance in day-to-day work, because of the effort invested in improving the code quality. It will take some time until these efforts are depreciated. Second, day-to-day work itself will be less efficient, because developers have to re-familiarize themselves with the improved system. It will take some time until the learning curve surpasses the performance level from the pre-change era. All this will make it easy for people resisting the change, to argue against it and endanger the software quality project.

In order to not let that happen the restructuring should take place in small steps, enabling short-term wins. To gain a short-term win, only the most relevant quality problems should be concentrated on first, in order to fix them as fast as possible. Most relevant problem does not necessarily mean generating the highest savings, but rather relevant in terms of convincing stakeholders. Let's suppose developers need to be convinced of the necessity of the change, and developers are constantly suffering under changing cloned code. A good candidate for a short-term win would be removing cloned code from the ERP system. Developers will feel the advantages of the improved code quickly and will be less likely to resist further improvements.

6.2 Comments on Consulting

When acting as a consultant to a company undertaking the transformation effort, short-term wins might also play the role of an initial project. The decision makers might not fully buy-in the advantages of software quality or not fully trust the consulting company to succeed in increasing performance through software quality, right from the beginning. In that case a short-term win project can help the consulting firm to convince decision makers and get bigger follow-up projects, which lead the company through a full-scale transformation.

7 Make Change Permanent

Now, that software quality was successfully implemented the question remains, whether the quality will be kept up in the future or whether it will degrade again. Under day-to-day pressure developers are in danger of switching back to "quick and dirty"-mode and managers might even foster this by demanding short development times.

7.1 Performance Metrics

The mechanism [3] suggests, in order to make the changed behavior of employees permanent, is performance metrics. His opinion that "employee allegiance is directed not to the company but to the location where they work" cannot be denied completely, but should be relativized considering his assumptions about workforce education.

However, if a company wants to reach high quality software, it makes sense that quality is also a parameter for measuring the performance of a developer, instead of just plain development time. To not corrupt this incentive by conflicting interests of middle management (like time pressure to finish project), quality need not only to be part of a developer's performance measure but also up the hierarchy. That way middle management will also have short-term interests in good quality software, they start supporting developers improving software quality, and become accountable for software quality.

For quality performance measures to work, developers first need to understand that the quality of their software is measured. Next, the performance measure needs to be communicated to them in a way they can relate to it. Intuitively understandable metrics such as "Percentage of Duplicated Code" should be preferred. However, some metrics are inherently not intuitively understandable, like manual inspections of samples. In this cases school grades can serve as a simple performance measure. Whatever measure is used, to make it easier for developers to accept the measurement, the estimation of the system quality should be reliable, objective, and sound. [6]

Important to note about performance measures is, that they are not a tool to convince people of a change, but to stabilize a change already in progress. If performance measures contradict a change effort, people will spend more thought on impeding the change in order to improve their metrics than on changing their behavior.

7.2 Involve more employees in change effort

After the first short-term wins are reached, the transformation should gain more and more support and more people getting involved in it. Therefore, bigger and more tedious transformation actions can be tackled. However, that means that the guiding coalition will not be able to control all transformation efforts in detail, any more.

That is, what [4] calls "leadership from senior management" and "project management and leadership from below". This involves following task for the guiding coalition:

– Maintaining a clear and shared understanding of where the transformation should lead to and how to get there
– Keep the urgency level up

For the latter point it is extremely helpful to not do "happy talk" like "we made the change" on annual executive meetings, in corporate newspapers, or elsewhere. The message should rather be, "we are on the right track but still have a long way to go". [4] emphasizes that making the changed environment the way the company does thinks, i.e. anchoring it in the culture, will in some cases take five or even ten years. It is not before then, that the organization will automatically stay 'transformed'. Before that point is reached, the wheel will automatically turn backwards if let go.

For middle management and group leaders "project management and leadership from below" means to also lead and manage sub-projects of the transformation. [4] makes clear that in order for this to work, the organization needs employees capable of leading. However, a common problem in transformation efforts is a lack of leadership in lower ranks of the hierarchy. [4] suggests systematical building of leadership in companies by flatter and leaner structures, broad based empowerment of employees, and a culture supporting more risk-taking.

7.3 Promotion and appointment of new employees

To firmly anchor the change [4] further suggests, paying attention to promotion and employment criteria. Having those set up to support the transformation, will make employees support the change in the long run, and will bring new employees into the company if their attitudes match the ones needed to support a changing company. New employee orientations can further improve the positive effect of new employees on the change culture [5].

8 Conclusion

This paper described how change management helps organizations to make the change towards software quality. The steps suggested are the following:

- Convince Stakeholders by Establishing a Sense of Urgency
- Create a Guiding Coalition
- Communicate Change
- Establish Short-term Wins
- Make Change Permanent

Software quality is not just a matter of introducing tools and frameworks. It is about changing the way software is developed. A tool cannot produce software quality; it can only support developers in a behavior that results in better software quality. In order to change employees' behavior, change management is crucial.

9 Further Topics

This report covered the organizational aspects of change management. When really introducing a change to an organization, a more detailed level needs to be considered as well. The exact wording that should be used or how group dynamics can help or impede to realize the change, are important questions that rather touch sociology and psychology research. "People change what they do less because they are given analysis that shifts their thinking than because they are shown a truth that influences their feelings" [5].

References

1. Kotter, J.: Leading change: why transformation efforts fail. Harvard Business Review (1995) 59–67
2. Lewin, K.: Frontiers in group dynamics: Concept, method and reality in social science; social equilibria and social change. Human Relations **1**(1) (1947) 5
3. Larkin, T., Larkin, S.: Communicating Change: How to Win Employee Support for New Business Directions. McGraw-Hill (1994)
4. Kotter, J.: Leading Change. Harvard Business School Press (1996)
5. Kotter, J., Cohen, D.: The Heart of Change: Real-Life Stories of How People Change Their Organizations. Harvard Business School Press (2002)
6. Mas y Parareda, B., Streit, J.: Software quality modelling put into practice. Technische Universität München, Technical Report TUM-I0811 (2008) Workshop-Band Software-Qualitätsmodellierung und -bewertung (SQMB '08).
7. Kiely, T.: Managing change: why reengineering projects fail. Harvard Business Review **73**(2) (1995) 15
8. Bungard, W., Fleischer, J., Nohr, H., Spath, D., Zahn, E.: Customer knowledge management: Erste ergebnisse des projektes customer knowledge management - integration und nutzung von kundenwissen zur steigerung der innovationskraft. Stuttgart: IRB-Verlag (2003)
9. Covington, J.: Eight steps to sustainable change. Industrial Management-Chicago Then Atlanta (2002) 8–11