COURSE BOOK



## Quality Assurance in Software Processes

IQSS01

Learning Objectives

##### Introduction 9



Quality assurance is one of the activities that accompanies a software process. The artifacts (documents, models, program code) created must be quality-assured from the beginning, since the cost of correcting an error increases if it is detected in a system at a later stage. In practice, however, industrial information systems cannot be fully tested.

This course teaches techniques and approaches for quality planning, quality assurance, and quality control. In addition to constructive measures for quality assurance, techniques for static and dynamic quality assurance are also taught. This includes activities for the systematic testing of software systems, as well as the quality assurance of requirements, documentation, architectures, and the software process itself.



# Unit 1

## Introduction to Software Quality Assurance

#### STUDY GOALS

After completing this unit, students will know ...

... what the most important terms in the field of software quality assurance mean.

... what the principles of software quality assurance and the fundamentals of software testing are, as well as their implications.

... why delivering error-free industrial information systems is generally not economically viable.

DL-D-IQSS01–L01

1. Introduction to Software Quality Assurance

### Introduction

This unit provides an introduction to the topic of software quality assurance. First, the most significant terms related to quality management are introduced. Then, the seven principles of software quality assurance are presented and their effects on QA activities are described. Subsequently, the fundamentals of software testing are explained prior to considering the topic of costs in relation to quality in greater detail.

### Motivation and Terms

Although it is only after its completion that a software system can be fully tested to ensure that it has met the stipulated requirements, the risk of only conducting quality assurance for a software process once it has been implemented is too high. For this reason, quality assurance in software engineering is carried out in parallel with all other accompanying activities.

###### Software Quality

A very general definition of the term *software quality* is provided in DIN ISO 9126 as follows:

“Software quality is the set of characteristics and characteristic values of a software product that relate to its ability to meet specified needs.”

In line with this definition, software must be reviewed to determine whether the product as delivered meets the requirements (needs) previously specified. The importance of a speciﬁcation is clear: according to DIN, the quality of software can only be determined on the basis of the speciﬁcation (“specified needs”). In practice, however, it is apparent that the perceived quality of a piece of software is primarily determined by whether the customer finds that it meets their actual requirements. Since many requirements are often only recognized during software development,care must be taken to ensure that the set of specified requirements also meets the actual requirements necessary to ensure customer satisfaction.

In addition to determining the quality of a software system as a whole, it is also possible to determine the quality of the artifacts created in the course of software development (software models, requirements, architectures, documents, software components).

Introduction to Software Quality Assurance

###### Quality Management

**Quality management** (QM) is a collective term that includes all organized measures that serve to improve the quality of products, processes, or services of any kind. Since it is too risky in software engineering to wait until the implementation is complete before determining the quality of the software as a whole, the quality of software fragments that have already been created is determined during the development process. Indeed, software quality management does not solely relate to the program code that has been created. Because the implementation is directly dependent on the specifications and decisions made in the course of determining the engineering requirements, specifications and architecture, errors in the specification or architecture definition can directly propagate to the implementation. Hence, these artifacts must also form part of quality management in software engineering.

Typical quality management activities include the following:

* Quality planning: preparation and documentation of quality requirements in collaboration with the client.
* Quality control: monitoring, management, and control of quality testing activities in the software development process.
* Quality assurance (QA): activities that ensure that the quality requirements specified for products, processes, and services are met.
* Quality improvement: evaluation of product and process data to improve the quality level.

###### Constructive and Analytical Quality Management

Fundamentally, quality management can be divided into the areas of constructive and analytical quality management.

In **constructive quality management**, all the quality attributes for products or processes are deﬁned *a priori* (i.e., before creation) in order to prevent errors during software development and to ensure or increase the quality of the artifacts created. The measures for constructive quality management include the following:

* Technical measures (e.g., use of modeling languages, tools, development environments),
* Organizational measures (e.g., guidelines, standards, templates, checklists), and
* Interpersonal measures (e.g., training, work climate, joint activities).

Quality Management

Quality management refers to all organized measures that serve to improve the quality of products, processes, or services: quality planning, control, assurance, and improvement.

Constructive Quality Management Constructive quality management consists of *a priori* measures to prevent errors before they occur.

Analytical Quality Management

Analytical quality management consists of measures for examining and evaluating the quality level of

test objects.

In **analytical quality management**, *ex post* (i.e., after creation) measures are carried out to examine and evaluate the current quality level of the test objects in order to systematically detect errors and determine their extent. Both static and dynamic testing procedures are used for this purpose, the most important attributes of which are compared in Figure 1 below.

|  |  |  |
| --- | --- | --- |
| Figure 1: Comparison of Static and Dynamic Procedures | | |
|  | Analytical Measures | |
| Static (analyzing) Procedures | Dynamic (testing)  Procedures |
| Method | Test object is only examined, but not executed. | Executable software is executed (run) with specific input values. |
| Test Object | Software development artifacts and processes, e.g., speciﬁcations, source code, architecture description, test documents, guidelines. | Executable program  and individual components |
| Examples | * Review of user manual for completeness. * Examination of whether the programming guidelines have been complied with. | Execution of the program with specific input values and comparison of the outputs generated with the expected outputs. |
| Examples of Testing Procedures | Review, metrics, static code analysis | Function-oriented tests, random tests |
| Positioning in Quality Management | Quality assurance, partially also quality control | Quality assurance |

In static procedures (also: static tests), the test object (a piece of software) is analyzed, assessed, and investigated as part of a review. The information obtained in the procedure is compiled, and if necessary, condensed into metrics or key figures, before ultimately being evaluated. In dynamic procedures (also: dynamic tests), the test object is executed with specific input values and the result of this execution is evaluated.

Introduction to Software Quality Assurance

Anticipatory constructive quality management can reduce the amount of work involved in analytical quality management.

###### Test Object

Fundamentally, all artifacts created within software engineering can be tested for compliance with quality requirements. This means that, in addition to the program code created,

* business (functional) requirements,
* technical speciﬁcations,
* architectural descriptions, and the
* test cases themselves

can also be tested. However, dynamic testing is only possible for executable artifacts. All other artifacts are tested using static testing procedures.

### Principles of SW Quality Assurance

Independent of specific techniques or approaches, Balzert (1998, p. 284–293) describes the following principles of software quality assurance that serve as the fundamentals to be considered during the specific design of QA activities and QA processes.

###### Principle of Product and Process-Dependent Quality Objective Determination

The principle of product and process-dependent quality objective determination means that the specific quality objectives that are relevant to the software product or software process should be determined in the early phases of the project. Documented quality objectives must exist so they can be considered and addressed later by the architects and developers during implementation.

In industrial practice, however, it is often the case that the targeted *a priori* determination of specific quality objectives does not occur. This is not least due to the fact that people often lack the necessary skills for the methodical determination and speciﬁcation of quality objectives on the basis of deﬁned quality models. Since the potential set of approaches and techniques for quality assurance is extensive, the required attributes can only be implemented in a targeted fashion once quality objectives have been formulated.

###### Principle of Quantitative Quality Assurance

The principle of quantitative quality assurance means that assertions regarding the quality of a system or process are made on the basis of key figures (also: metrics) for which there are specified target values. The actual values measured are compared with the target values.

Example

“On the first release of the application, all known critical errors, but only 70% of all known non-critical errors, must be fixed.”

While the measurement of attributes has proven to be practical, there are no standard metrics that are used for every software process. The values that are actually measured depend on the organizational structure of the project, the functional environment, and the requirements of the product and the software process.

###### Principle of Maximum Constructive Quality Assurance

The principle of maximum constructive quality assurance means that, in quality management, all measures that contribute to the prevention of errors should be taken. The assumption of constructive quality assurance is that high product quality can be achieved through high process quality.

###### Principle of Early Error Detection and Correction

The principle of early error detection and correction is intended to ensure that errors are detected even before they manifest themselves in the program code. Requirements engineering (RE), as well as the speciﬁcation and creation of an architecture deﬁnition, occur before the actual implementation begins. Errors in RE propagate through the speciﬁcation and architecture deﬁnition and into the implementation. The later an error is identiﬁed, the higher the cost of correcting the error. If an error is not discovered until a system is delivered, it may be necessary to go through all the phases up to implementation in order to correct it.

However, if, for example, an error is discovered in the requirements document before the speciﬁcation is drawn up, only the affected parts of the requirements documentation need to be adjusted. Correcting errors in early phases is therefore much quicker and more cost-effective than correcting errors that extend from the business requirements document right through to implementation. In practice, however, a large proportion of the errors caused in early phases are not detected until the acceptance test.

Introduction to Software Quality Assurance

To prevent this, structured QA measures should not begin when the software is tested, but rather when the first artifacts are created in a software process.

###### Principle of Development-Accompanying, Integrated Quality Assurance

This principle is the consequence of the principle of early error detection. In order to detect errors as early as possible, quality assurance activities must accompany development activities and be firmly integrated into the software process. Consistent application of this principle requires quality assurance to be defined for all artifacts created in the software process to ensure that errors do not propagate from one activity to the next. This principle is implemented particularly succinctly by the evolutionary development process model. Here, the software is developed over several iterations, the result of which includes testing the software each time. In this way, errors can be detected in good time and their propagation into all subsequent iterations can be prevented.

###### Principle of Independent Quality Assurance

The principle of independent quality assurance states that the review of the artifacts created should be as objective as possible, without being impacted by individual interests. For example, if a program code is to be tested by the team that creates the code, the time allocated for QA is often not used for QA to the extent required. Instead, they use it to implement additional functions. However, if QA is completely removed from the development team’s responsibility, they are often observed to take less care during the development stages, since others are responsible for finding errors.

In practice, a model is often used that takes both aspects into account: the development team is responsible for ensuring the quality of its results up to a quantitatively defined level. Then, the quality assurance team examines all the aspects that go beyond that level. For example, the development team must ensure that automatic test cases exist for every function implemented and that they all run successfully. In addition, particularly critical code sections must have undergone a manual review. Based on this, the quality assurance team tests the functional use cases, i.e. the correct interaction of all development artifacts.

### Principles in Software Testing

In addition to the principles of software quality assurance, targeted fundamental attributes and phenomena relating to software testing are described. For this purpose, Spillner/Linz (2019, p. 22f) present the following principles in software testing, among other aspects.

###### Testing Shows the Presence of Errors

Errors in a system can be identiﬁed by conducting tests, which means that existing errors can be found. However, even if all the tests are successful, no assertion can be made about the absence of errors. Extensive testing reduces the probability that undetected errors still exist, but an absence of errors cannot be proven by software testing.

###### Complete Testing is Not Possible

Complete testing is not possible, particularly with industrial information systems. The quantity of possible input values and the quantity of possible system states are so large for industrial software that a complete testing is either technically impossible or would require so many resources in practical execution that the system to be tested could never be operated economically. Therefore, the software tests actually conducted are always only random samples. The decision as to which functions are tested with which effort and expense, as well as which input parameters, must therefore be made consciously and very carefully.

###### Accumulation of Errors

The errors detected in software systems are usually not evenly distributed across all system components. The majority of errors are often concentrated in just a few components. Thus, if an error is detected at location X in the system through testing, it is

Introduction to Software Quality Assurance

highly probable that further errors are present in the environment of X. The phenomenon of error accumulation must therefore be considered when conducting software tests. Testing must be conducted with particular care in an environment where errors have already been identiﬁed and, if necessary, using more test cases than originally planned.

###### Repetition Has No Effectiveness

The repeated execution of the same test cases with the same input data in the same system state and on the basis of the same data set without modifying the program code will not achieve any effect different from that generated by executing the test cases once. The repeated execution of the same test under identical conditions will therefore not lead to gaining additional knowledge regarding potential errors.

###### Testing Depends on the Environment

In line with the above-mentioned principle of product and process-dependent quality assurance, the number and type of test cases are newly determined and formulated for each system. All activities—from conception and creation to the implementation and evaluation of test cases—are unique for each project. They are determined on the basis of the requirements for the system, the boundary conditions of the project and the quality objectives that have been formulated. Strictly speaking, it is a “software project within a software project” that runs through all phases from the requirements analysis to implementation, particularly when using automatic software tests.

###### Fallacy: No Errors Does Not Automatically Mean a Usable System

The quality of software, and thus also the formulation of test cases, is based on the speciﬁcation of the system. Aspects that may be important to users, but were not considered during the speciﬁcation, are usually not tested. Therefore, even a very thoroughly tested system in which all the detected errors have been removed may be unusable from the user’s point of view. If, for example, a system becomes very slow in its operating environment as part of the application landscape with a high number of concurrent users or is only available to a limited extent, there is no software error in the sense of a faulty system function, but the expected quality attributes are not met. This, in turn, affects the system’s usability and its perceived quality. This principle can be summarized with the following quote by the computer science pioneer Edsger Dijkstra:

“Program testing can be used to show the presence of bugs, but never to show their absence!”

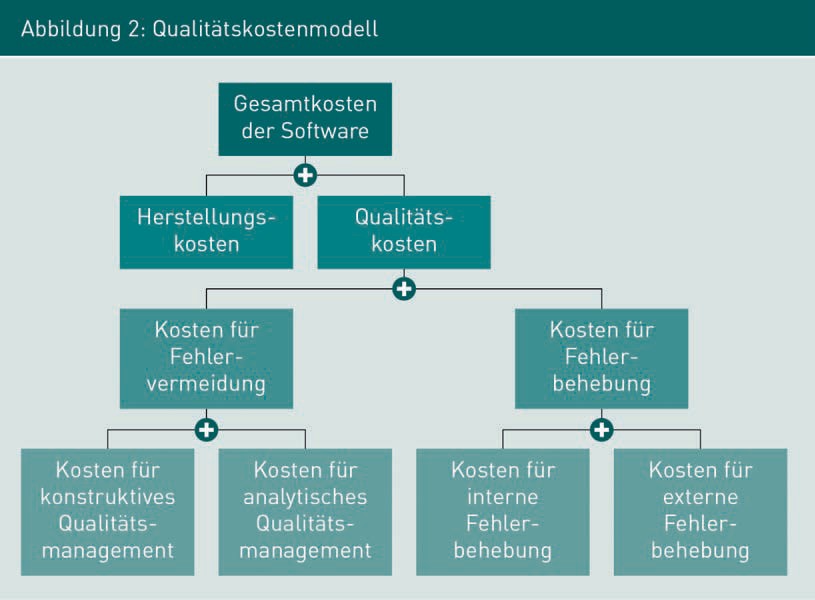
Phenomenon of Error Accumulation

Errors are usually not evenly distributed. If an error is detected at location X in the system through testing, it is highly probable that additional errors are present in the environment of X.

### Quality Costs

As already described above, software tests can neither be conducted completely, nor can they prove that software is free of errors. An individual decision regarding the extent of resources to be spent on quality management must be made for each software project. The quality cost model shown in Figure 2 below, which is based on the model presented by Grechenig (2010, p. 468–470), is used for assessment and decision-making.

The total costs of a software system are determined by the costs of its production and quality control Production costs are the costs incurred by producing the desired functions and the desired quality characteristics, while quality costs include the effort and expense applied to error prevention, detection, localization, and correction.

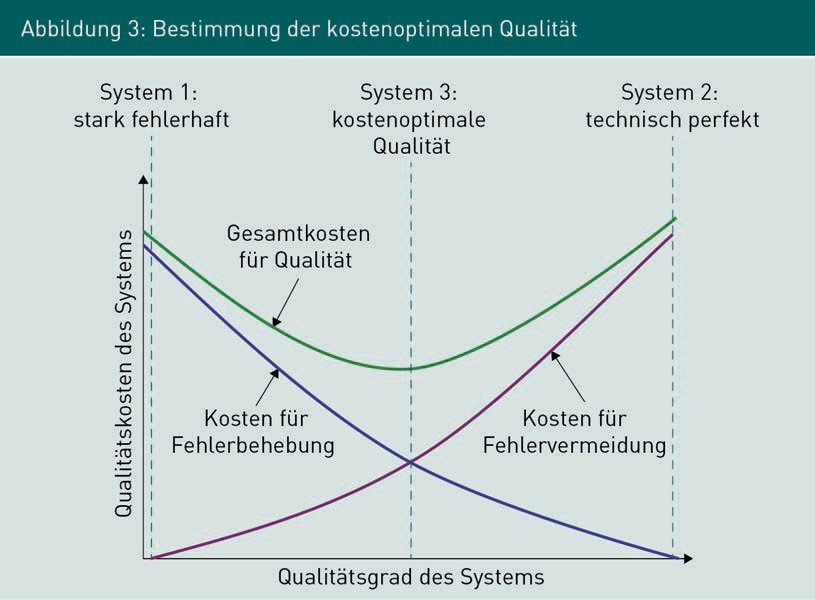


In specific terms, quality costs consist of the costs for error prevention and the costs for error correction. Error prevention includes all activities that are conducted within the software process prior to delivery of the system with the objective of ensuring that as few errors as possible are discovered after the system has been delivered. The costs for error prevention therefore include all activities for constructive and analytical quality management within the software project.

Introduction to Software Quality Assurance

Costs for error correction are incurred when errors are discovered after the system has been delivered. The effort and expenses for localizing and correcting errors come into effect as internal costs for error correction. In particular, they include the effort and expense for software tests that must be conducted after modifications have been made to the source code. External costs can certainly also arise, for example, due to contractual penalties. Alongside this, damages that cannot be precisely quantified can often be expected, for instance, due to the customer losing confidence in the software manufacturer, or reputational damage being suffered by it.

In industrial practice, the decision as to how resources can be made available for error prevention must always be made from an economic point of view. Figure 3 below, in which the quality costs of a system are shown in relation to the quality level of a system, serves to illustrate these conflicting priorities.



The violet line illustrates the costs for error prevention—these are not linearly dependent on the degree of quality achieved. Up to a certain point in the software process, error prevention measures contribute relatively quickly to a higher degree of quality. However, the more frequently and the more intensively quality assurance is carried out, the more difficult it becomes to ﬁnd further errors. The effort and expense for quality assurance increase disproportionately in relation to the actual increase in the quality level. The blue line illustrates the costs for error correction. A largely untested system generates extremely high correction costs, while correction costs decrease more slowly with a higher quality level.

Figure 3 above shows three different system types with dashed lines as examples. System 1 is a highly faulty system. The costs for error prevention in System 1 are very low, since next-to-no quality management was carried out. However, the costs for error correction are very high, since errors in the system were not detected until after it was delivered. In contrast, the technically perfect System 2 is shown on the right. In this system, very high costs are incurred for error prevention measures, but almost no costs are incurred for error correction. In this case, almost all the errors were detected and corrected before delivery. System 3 is a system that was optimally created in terms of quality costs. The most significant errors were identiﬁed and corrected before delivery, but some errors were also identiﬁed and corrected after delivery. The costs for error prevention and error correction are the same for cost-optimized quality.

When planning within the context of quality management, the point of optimal quality costs in a project should be aimed for. On the one hand, any errors that are very expensive to correct afterwards should be prevented. On the other, the costs for preventing errors should not be greater than the costs for correcting these errors. For this purpose, the risks must be evaluated along with their probabilities of occurrence and consequential costs. Drawing on the results of the risk analysis and the available project budget, quality assurance measures must be planned in such a way that they can be implemented at the lowest possible cost.

Summary

Quality management (QM) is a collective term that includes all organized measures that serve to improve the quality of products, processes, or services of any kind. In constructive quality management, all quality attributes for products or processes are deﬁned before creation in order to prevent errors while the system is being created. In analytical quality management, the artifacts created in the software process are tested. Various principles and fundamentals are described for the activities within software quality assurance, which must be considered during specific quality planning and quality assurance.

In industrial practice, the decision as to what budget is available for error prevention is always made from an economic point of view. When carrying out quality management planning, the point of optimal quality costs in a project should be aimed for. The cost of preventing errors should not be greater than the costs for correcting them. The actual optimal effort and expense can only be determined if the costs for error correction are known.



# Unit 2

## Software Quality Organization and Planning

#### STUDY GOALS

After completing this unit, students will know ...

... how product-oriented and process-oriented quality management differ.

... what quality planning means and what role quality objectives play in it.

... which typical quality assurance activities can be planned in a software process.

... how total quality management (TQM) can be applied to quality control.

DL-D-IQSS01–L02

1. Software Quality Organization and Planning

### Introduction

The quality of a piece of software does not occur by chance. Rather, it must be systematically planned and organized. Active quality management is particularly necessary since quality assurance activities extend over the entire software process, while individual quality objectives apply to each unique project. This unit introduces the most important quality management activities before individual aspects are discussed in more detail in the following units.

### Overview of the Quality Management Process

###### Core Activities in Quality Management

The term *quality management* is deﬁned in the industry standard ISO 9000 as follows:

“Coordinated activities for managing and directing an organization with regard to quality, which usually includes establishing quality policy and quality objectives, quality planning, quality control, quality assurance, and quality improvement.” The activities required for quality management can—and must—also be carried out for each individual software process within the scope of software development. This is because the quality objectives must be redefined for each individual software process. Therefore, software quality management does not focus on the design of quality management for an entire organization, but rather on the activities and responsibilities within a specific software process.

For each specific software process, quality planning determines exactly which individuals are responsible for which QA activities, which QA activities are required at which point in time in the software process, and with which resources, methods, and tools quality assurance is carried out.

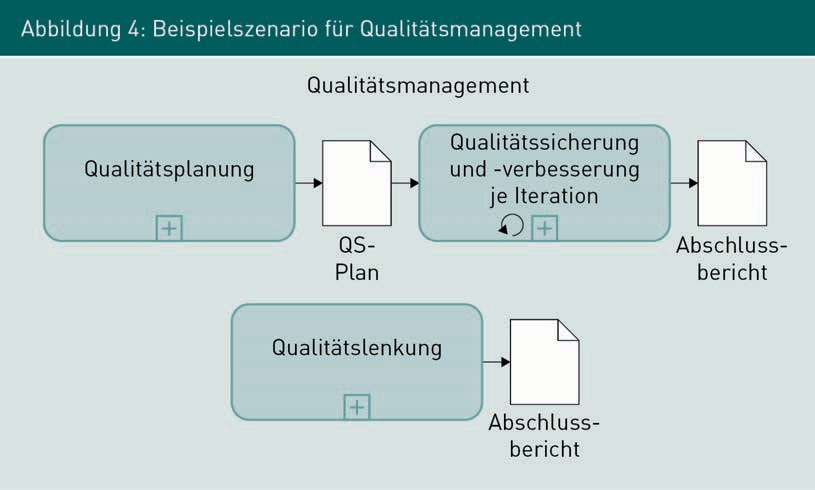
The measures determined in the quality plan are carried out during quality assurance activities. The quality plan therefore provides a specific framework for action within a project, so that every employee in the project knows at all times when which QA activities are due, how the results are documented, and who is responsible for carrying out the activities.

*Quality control* is a collective term that includes all the activities required for monitoring and controlling QA activities. Here, the focus is not on the software system, but rather on the activities that are carried out for quality assurance. The objective of quality control is to achieve an appropriate level of quality assurance.

Software Quality Organization and Planning

In software projects, quality improvement activities are often carried out in conjunction with quality assurance activities. In terms of the artifacts that typically result in the software process, quality improvement means reworking text, models, and program code. For this reason, quality improvement activities will not be discussed further as this course progresses.

Figure 4 below shows a specific example scenario of software quality management. The result of the quality planning activities is a quality assurance plan that includes precise QA measures. These are carried out as part of quality assurance for each software release and documented at the end of the project in the form of a final report. Quality improvement activities are also carried out for each release in parallel with quality assurance. This is accompanied by quality control activities, which are also documented at the end of the project in the form of a final report.



As already mentioned, it is not only the artifacts created in a software process that can be quality-assured, but also the software process itself.

###### Product-Oriented Software Quality Management

Product-oriented quality management reviews software products for specific quality characteristics as well as the intermediate results generated during the creation, maintenance, and further development of the software. In addition to the project-specific quality characteristics, application software is occasionally required to meet additional quality conditions that are tested and confirmed by external certiﬁcation bodies. Examples of this include the testing of applications published in the iTunes Appstore by Apple, or the TÜV AppCheck service for certifying mobile applications for compliance with data protection criteria.

Measures for product-oriented software quality management include constructive measures, such as the specification of certain document templates or the specification of modeling conventions, as well as analytical measures, such as the static assessment of source code or the running of executable components within the scope of software testing.

###### Process-Oriented Quality Management

Process-oriented quality management is used to control the quality of the software process. The focus is not on the artifacts created, but rather on the quality of the implementation of the activities planned in the software process and the use of the planned techniques and methods. Process-oriented quality management accompanies the software process and is characterized by a continuous evaluation and, if necessary, improvement of the process quality. An important prerequisite for effective process-oriented quality management is the quality awareness of the team members. It is only when the quality of the collaboration is critically reflected upon and examined on a regular basis that continuous process improvement is possible. Examples of measures for process-oriented quality management are the use of a standardized development process that specifies roles and activities to an appropriate degree, the mandatory use of tools for conﬁguration management, or the specification of escalation paths in the event of a project disaster.

### Quality Planning and Quality Objectives

During quality planning, all the specifications for the organization and implementation of quality assurance measures are defined in addition to the specific quality objectives of the system to be created. While considering the risk analysis and the available project budget, the quality assurance measures must be planned in such a way that they can be carried out at the most optimal cost.

Quality planning occurs in the early phases of the project. Quality objectives and quality indicators for the system to be created are defined in collaboration with the client and any other relevant stakeholders. In the course of this, the specific quality objectives are determined with the aid of a suitable quality model and corresponding quality indicators are derived for each quality objective.

Example:

Usability was defined as an important quality characteristic for an online store. The number of dialog screens displayed from the shopping cart to the completion of the order was identified as a measurable quality indicator. A specific quality requirement for the online store was formulated as follows: “The number of dialog screens displayed in the GUI (graphical user interface) for the order from the shopping cart to completion of the order must not exceed 2 for existing customers and 4 for new customers.”

Software Quality Organization and Planning

###### Planning the Organizational Achievement of Quality

In addition to determining quality objectives for the software system that has been created, the organizational achievement of quality through deﬁned activities, methods, and techniques in the software process is also defined during the course of quality planning.

The following aspects must be considered within the scope of organizational quality planning:

* Who is to assure quality? It must be clear at all times in the project which team members are assigned which roles with regard to quality management and which tasks are assigned to the area of responsibility assigned to the individual roles.
* What is to be quality-assured? For each result type created in the course of the software process, the criteria that are to be met for the artifact created must be determined so that the required quality level is present. To this end, measurable quality indicators must be determined and documented on the basis of the quality objectives that have been defined as relevant.
* When should quality be assured? The various QA activities must be correlated in a chronological relationship with the constructive activities in the software process. In addition to the targeted definition of activities, what are known as *quality gates* are often deﬁned. A quality gate is a determined point in time in the software process at which the deﬁned attributes of the artifacts at hand must be explicitly examined. If one or more required attributes are not met, the subsequent activities cannot begin.
* How is quality to be assured? During quality planning, specific techniques and methods for quality assurance or test case generation are determined. They are selected from the portfolio of suitable techniques based on those that can achieve cost-optimized system quality according to the current state of knowledge. In the course of the software process, the techniques and methods specified in the quality plan may need to be adapted to the current project situation.

The planning activities give rise to a project-specific quality assurance plan (also: quality management plan) in which all the responsibilities, results, activities, and methods are documented. Figure 5 below shows typical activities that are carried out as part of comprehensive quality planning for a software project. The specific planning activities always depend on the type and size of a project, as well as the specific project organization. In general, they are not processed in a strictly sequential fashion, as illustrated in Figure 5. Rather, this illustration serves as an overview and checklist to determine whether all the important points have been taken into account during quality planning.



All activities shown in Figure 5 are described in Table 1.

|  |  |
| --- | --- |
| Table 1: Quality Planning Activities | |
| Activity | Description |
| Define responsibilities and fill roles | Designation and description of all roles that become active in QA activities, as well as assignment of the roles to specific individuals. Description of the organizational structure of the project, if necessary. |
| Establish committees and regular meetings | Designation of committees and description of their responsibilities; establishment of regular meetings of these committees. |
| Define and prioritize quality objectives | Based on a quality model (e.g., ISO 9126), definition and then prioritization of the quality objectives relevant to the project. This result serves as one of the most important guidelines for all further activities. In general, coordination with the client is necessary. |

Software Quality Organization and Planning

|  |  |
| --- | --- |
| Activity | Description |
| Define QA activities and quality gates in the software process | Specific QA activities such as reviews or software testing are defined and positioned in relation to the constructive activities in the software process. In addition, quality gates are used to integrate very important decision points into the software process. It is at these points that decisions are made regarding the further course of the project on the basis of the measured quality. |
| Define acceptance criteria and committees for important result types | In order to ensure that all artifacts created in the project are of the desired quality level, acceptance criteria are defined for important artifacts (such as business requirements, speciﬁcations, source code, test cases, planning documents, etc.). Additionally, the group of people to be involved in preparing the acceptance and the ofﬁcial acceptance is defined. |
| Define constructive QA measures | Constructive measures include all specifications and guidelines that are defined for the activities in the software process in order to prevent the occurrence of errors. Examples of this are checklists, templates, programming guidelines, specification of determined tools, methods, techniques, and procedures. |
| Define analytical QA measures:  static procedures | Description of the processes and results of static test procedures such as reviews or static code analysis. Since, in principle, all created artifacts can be statically tested, it may be necessary to differentiate between them based on artifact type. |
| Define analytical QA measures:  dynamic procedures | Description of the processes and results of dynamic test procedures, typically for each relevant test level. This also specifies how test cases and test data are to be generated. |

|  |  |
| --- | --- |
| Activity | Description |
| Define testing infrastructure | Since industrial information systems are usually operated in a complex application landscape, are connected to other systems via technical interfaces, and often process and store many hundreds of thousands of data sets, the system must run in an environment that is as close to the production environment as possible, depending on the test level. Therefore, the provision of testing infrastructures is often very costly, albeit indispensable for integration testing and system testing. |
| Define build processes | The code fragments created by the individual developers are compiled into an executable system by what is known as a build process.  Automatic software tests can be conducted during compilation. When a build process is initiated, what type of tests are conducted and how to react in the event of an error is defined in the quality plan. |

###### Quality Objectives

By defining and prioritizing quality objectives, project-wide guidelines are created to which all QA activities must be oriented. In principle, the quality objectives relate to the individual QA activities in the same way as the project vision relates to the constructive activities: quality objectives form the point of reference to which the individual QA activities must orient themselves. Quality models can help make the abstract concept of quality tangible. Software quality models, in particular, are used to make software quality tangible in this way. The basic idea of a quality model lies in breaking down the abstract, broad term *quality* into various sub-terms. In turn, these sub-terms are broken down until a specific measurable quality attribute can be formulated. A quality model therefore consists of a hierarchy of terms that can help when specifying tangible quality attributes. Figure 6 below shows a quality model according to ISO 9126 as an example. The term *software quality* is divided into six characteristics that are each further divided into sub-characteristics.

Software Quality Organization and Planning



The SMART model from the fields of project management and personnel development can be used as a tool for reviewing whether quality characteristics have been suitably described. Both are areas in which objectives must be agreed and their achievement examined. Therefore, the quality objectives that have been formulated can also be described as SMART. SMART is an acronym which, according to Schiersmann (2000, p. 169ff.), can be broken down as follows:

* S = speciﬁc: quality objectives should be clearly described and deﬁned.
* M = measurable (qualitative, quantitative): it must be possible to examine the implementation of the quality objectives.
* A = acceptable: the quality objectives must be accepted by the project stakeholders; this also includes the acceptance of the testing methods.
* R = realistic: it must be possible to achieve the quality objectives, taking into account the professional and organizational framework conditions.
* T = time-bound: the quality objectives must be achieved within a specified time frame,

e.g., in a defined release.

In practice, the specific quality objectives for a project are derived from various sources. For example, large companies often have their own quality model that is oriented to the speciﬁc needs of the organization. Furthermore, there are also organization-specific QM manuals or an IT strategy that can be used as guidelines for the definition

of quality objectives. In addition, the specific goals or vision of the project are a source of quality objectives, as are the client’s contractual specifications. Stakeholders’ personal experiences with comparable projects or systems, particularly those of the quality manager, are also sources of quality objectives that must be taken into account.

In general, a large number of relevant quality objectives are identified in early project phases. Therefore, these objectives must be prioritized so that the individual QA activities can take this priority ranking into account in the further course of the project. In addition, care must be taken to ensure that the quantity of quality objectives identified is also quality-assured. Important objectives cannot be achieved if they are not identified and thus not documented. In contrast, excessive or incorrectly formulated quality objectives involve the risk of incurring effort and expense for achieving these objectives that do not benefit the system’s customers.

### Quality Assurance and Quality Improvement

The individual activities for quality assurance and improvement are generally defined according to the individual organization’s specifications, but are also specifically defined for each individual project by the project management, in conjunction with the quality manager. Typical examples of specific activities for quality assurance and improvement in a software process are provided and explained using the example of the quality plan shown in Figure 7 below. The main constructive activities of the software process (SW Development) are shown on the left-hand side of Figure 7, and the main quality assurance activities are shown on the right. The elements that explicitly contribute to quality assurance are marked with a red asterisk (\*). In addition to the various activities and their results, Figure 7 also contains what are known as quality gates. These are shown between both SW development and quality assurance pools. Revision cycles are not shown in this instance, but are required in the event that tests or quality gates are not passed.

\*K: Constructive Quality Management

Defined templates are to be used for all elements marked with *\*K*—these are all result types in Figure 7. In addition, there is a responsible role and a deﬁned acceptance process for each document. There are specifications for activities marked with \*K: there are templates and checklists in requirements engineering and architecture. For implementation, there are specified tools and programming languages; for the test case generation activity, there are guidelines for test data determination; for the test phase activities, there are guidelines for the use of test automation tools and for regression capability.

\*1: Examination of the Business Requirements

Examining the business requirements ensures that the documented requirements are of sufficient quality with regard to the criteria of content, documentation, and consistency.

Software Quality Organization and Planning

\*2: Quality Gates

The quality plan shown in Figure 7 above provides for a total of six quality gates, five of which must be passed in each iteration. The objective of quality gates is to ensure that the artifacts created fulfill previously specified attributes at a certain point in the software process. Quality gates are also often used to ensure that quality development is transparent and traceable. A quality gate can be passed if the previously specified criteria, e.g., in the form of a checklist, have been met. In addition to the quality of the artifacts created, quality gates are often used to check compliance with specifications and guidelines for the software process. Table 2 below describes the quality gates shown in the example.

|  |  |
| --- | --- |
| Table 2: Examples of quality gates in the software process | |
| Quality Gate (QG) | Description |
| QG 1: Examined requirements | A technical speciﬁcation may only be carried out when examined requirements are available. The requirements are ofﬁcially approved. |
| QG 2: Examined specifications | The technical speciﬁcation must first be examined and approved before the creation of the architecture description and the test case generation may begin. |
| QG 3: Suitable architecture | The architectural description is created and the decisions made in the process were positively evaluated as being suitable for meeting the requirements. |
| QG 4: Tested components | The components created (or modified) by the development team have been implemented and tested at the module and component test level. |
| QG 5: Tested system | The system test is complete and all errors relevant to the current iteration have been corrected. |
| QG 6: Accepted system | The system has been accepted by the customer and can be put into operation. |



Software Quality Organization and Planning

\*3: Examination of the Technical Speciﬁcation

Examining the technical speciﬁcation ensures that the system components to be implemented are described to a scope and level of detail appropriate for the architecture and implementation.

\*4: Evaluation of the Planned Architecture

The architecture description created and the decisions made regarding it are evaluated, for example, with scenario-based architecture analysis (see Unit 7). For this purpose, specific business application scenarios derived from the speciﬁcation are used to analyze whether the architecture design can fulfill the requirements.

\*5: Test Case Generation

The generation of test cases for the integration and system testing can be started in parallel to the architecture creation based on the business requirements and the technical speciﬁcation.

\*6: Component Tests

Software tests are carried out in parallel and closely interlinked with the implementation phase. This involves component tests in which components and functions created by a developer are tested in isolation from other components. In addition, such tests are also performed at this stage to review the interconnection of all components created in the current iteration.

\*7: Integration Test

Once implementation is complete, newly created and modified components must be integrated into the overall system. The integration test is a business end-to-end test that examines the interaction of all the components involved in the system. Automatic GUI regression tests are used to ensure that successful tests from previous iterations are also successfully run again.

\*8: System Test

The system test is performed in an environment that corresponds as closely as possible to the subsequent operating environment or at least comes very close to it. In particular, the system test examines system behavior in extreme situations, such as simultaneous access by many users, its behavior when processing large amounts of data, its reaction in the event of the failure of individual components, and its behavior in interaction with other production systems.

\*9: Acceptance Test

The acceptance test is conducted by the customer and usually corresponds to a subset of the system tests carried out by the software manufacturer during the testing phase. Based on the result of the acceptance test, the customer decides whether the system meets the requirements or whether a rework is necessary.

### Quality Control

Quality control activities (also: quality management, quality regulation) serve to improve the quality of QA activities. Quality control is needed because of the connection between the quality of QA activities and the quality of the product. Assuming that the quality level of software systems depends on the quality of QA, QA activities must be actively controlled in industrial software engineering. In this context, control means the continuous observation of process and product quality and the reaction to changes in the quality level in the form of adjustments to the QA specifications, as well as the testing of the actual QA quality. The findings from quality control activities influence quality planning specifications and guidelines, which, in turn, define specific QA activities, techniques, and methods for the software process.

The concept of total quality management (TQM) is presented below as an example of a fundamental approach to quality control (see Balzert 1998, p. 340–355).

###### Total Quality Management (TQM)

Compared to traditional approaches to quality management, in which responsibility for quality control is assigned to specific roles, TQM explicitly involves all of an organization’s employees in quality control. In TQM, the achievement of high product quality is the overarching objective pursued by all the employees within an organization. The customer alone decides on the level of quality achieved. TQM is based on the assumption that complete and continuous concentration on high quality ensures long-term business success through customer satisfaction.

Total quality management is not based on a standard process or standard activities. Rather, it is a management principle for the design of organizations and processes. The following principles can be described for TQM and are suitable as a basis for designing one’s own processes and guidelines:

* + Principle of customer orientation
  + Principle of quality priority
  + Principle of responsibility of all employees
  + Principle of continuous improvement
  + Principle of internal customer-supplier relationship
  + Principle of process orientation

Software Quality Organization and Planning

Principle of customer orientation

Satisfied customers are the primary objective, since the customer's perception determines the quality of the system. Therefore, the development team must know the actual requirements of the customer and the benefits of the system for the customer. In general, this requires support from the customer in formulating the actual requirements.

Principle of quality priority

All processes are carried out exactly as they are deﬁned. Each person involved in the execution of a process performs their tasks correctly from the beginning—as well as with each repetition. Reworking and wasting resources should be avoided. If errors are identiﬁed, their cause must be determined and eliminated. Quality improvements are achieved by optimizing the defined software process.

Principle of responsibility of all employees

Every employee involved in a project is jointly responsible for achieving an appropriate level of quality. Managers must enable employees to work while making as few errors as possible and manage all processes from the point of view of process and product quality. For example, responsibility for this cannot be delegated to a QA department.

Principle of continuous improvement

In general, organizations never run perfectly, and by extension, neither do software processes. The principle of continuous improvement means that process quality should be continually improved through small, but continuous steps. To this end, weak points and suggestions for improvement can and must be identiﬁed and communicated by all employees throughout the process.

Principle of internal customer-supplier relationship

Internal service provision is organized in a similar way to an external customer-supplier relationship. Services rendered are accepted and handed over as would be the case for an external customer. Each team is responsible for the quality of its own services. The performance of a team is measured by the satisfaction of internal customers.

Principle of process orientation

Quality deficiencies and errors are primarily seen as weak points in the development process that are to be identified and eliminated. The review of a product serves as a review of the process quality.

Summary

Product-oriented quality management can be fundamentally distinguished from process-oriented quality management. Product-oriented quality management both reviews software products for specified quality characteristics and assesses the intermediate results generated during the creation, maintenance, and further development of the software. In contrast, process-oriented quality management serves to control the quality of the software process.

Quality planning activities use quality models to define the specific quality objectives of the system to be created, as well as all specifications for the organization and implementation of quality assurance measures. The objective is to achieve the best possible quality at the lowest possible cost. The specific activities for quality assurance and improvement are usually defined for each individual project by the project management and the quality manager.

Within the scope of quality control, process and product quality are continuously monitored and changes in the level of quality are responded to in the form of adjustments to the QA specifications. The concept of total quality management (TQM) is an example of a fundamental approach to quality control.



# Unit 3

## Constructive Quality Management

#### STUDY GOALS

After completing this unit, students will know ...

... which activities and measures can be assigned to constructive quality management.

... how the 5-why method is used for root cause analysis.

... what timeboxing means and how checklists can be used in constructive quality management.

DL-D-IQSS01–L03

1. Constructive Quality Management

### Introduction

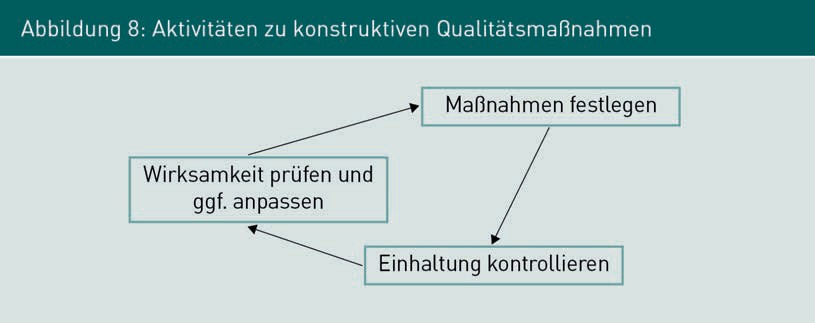
Measures for constructive quality management are taken to ensure high process quality and to prevent errors before they even occur. In this process, specifications are defined that are to be taken into account during the handling of specific activities within the software process. The fundamental concept behind constructive quality management is that the quality of a product depends on the quality of the creation process.

### Overview of Constructive Quality Assurance

Constructive quality management measures serve to ensure the quality of the design process, with binding specifications and guidelines intended to prevent errors from occurring. For example, a deﬁned software process itself counts as a measure for constructive quality assurance. In contrast to analytical QM, created artifacts are not taken into account as part of constructive QM.

###### Activities within Constructive Quality Management

The definition of the constructive QA measures is part of the quality planning, as shown in Figure 5 above. Figure 8 below provides an overview of the activities for constructive quality measures. The measures that are considered to be helpful for a specific project to achieve cost-optimized quality are selected from the set made available. It should be noted that the specifications for the project not only eliminate error symptoms: they also remove the actual cause of the error. A technique known as *root cause analysis* is frequently suitable for this.

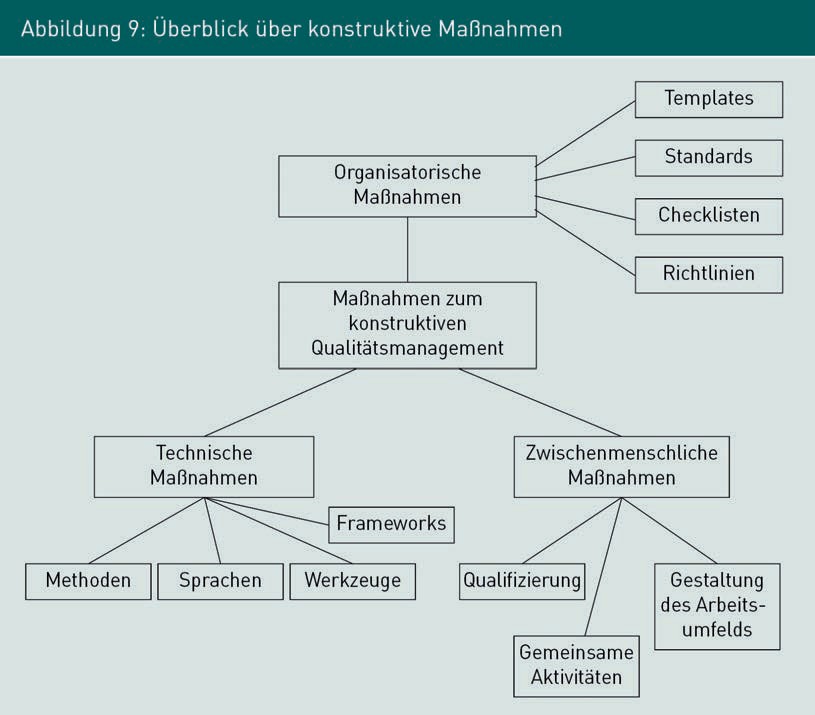


Constructive Quality Management

In addition to defining the measures, compliance with them must be monitored subsequently during the software process and explicitly enforced, if necessary. Compliance with the specifications is examined either in the testing activities for each artifact or when deciding whether a quality gate can be passed. In addition, the effectiveness and quality of the specifications must be examined regularly and the specifications adjusted, if necessary.

###### Measures for Constructive Quality Management

Measures for constructive quality management in software engineering are aimed at preventing errors during activities for the design, operation, and further development of software systems. Figure 9 below provides an overview of measures for constructive QM, which can be categorized into organizational, technical, and interpersonal measures.



Organizational Measures

**Organizational measures** include all non-technical aspects that have an impact on the specific organization of work. They include the specification of standards, guidelines, and checklists that are to be complied with. For example, standards can be industrial standards (such as ISO 9000, ISO 9126), widely recognized process models,

Organizational Measures

These are measures that have an impact on the specific organization of work and include all non-technical aspects (e.g., standards, templates, guidelines, and

checklists).

(such as RUP, V-Model XT, Scrum), or internal organizational standards (such as contract award regulations). Templates are generally company or project-specific templates for document structures and content. Templates can be created and prescribed for result artifacts (such as requirements, speciﬁcations, documentation, source code), as well as for management artifacts (such as status reports, final reports, error messages). A possible scope for decision-making can be specified through the use of guidelines. Decisions made during the implementation of the project should adhere to the specified guidelines. Typical examples of guidelines in software engineering include programming guidelines, behavior guidelines, and conventions. Timeboxing is a guideline for the organization and conducting of meetings. Checklists support the achievement of a constant level of quality for activities that cannot be meaningfully described by a process deﬁnition. For example, a checklist contains all the activities to be completed for a task and/or the required result or management artifacts and is typically created for repeated standard tasks, such as briefing new employees, performing document reviews, or examining the current process step at quality gates.

Technical Measures

*Technical measures* is a collective term that includes all the aspects that relate to specific (software) technical aspects in the software process, i.e., the specifications of methods, languages, tools, and frameworks from industrial software engineering.

Software engineering methods are techniques and approaches that can be used to complete specific tasks in a structured and systematic manner. Examples of methods are ranking (for prioritization), functional equivalence partitioning (for test case generation), and complete user stories (for requirements engineering). The most important result artifact of a software project is the software system, which is programmed in one or more programming languages.

In general, the choice of languages to be used in the project is therefore specified. In addition to the programming languages for the program logic (such as Java, C#, JavaScript, or Python), the programming languages for very specific components (such as JavaScript for the GUI, SQL for databases) are also determined, as are modeling languages (such as UML, E/R diagrams, BMPN, ARIS, Data-flow diagram). The determination of the language also includes the determination of the specific language version or possible dialects or special features to be considered when using it. In addition to the methods and languages, precise specifications for the tools to be used are also defined. Tools are software systems that support the activities in a software process. They include development environments (such as Netbeans, eclipse, Visual Studio), testing tools (such as JUnit, Selenium), and systems to support project management and communication (such as Microsoft Project, Microsoft Teams, Github, Gitlab, Bitbucket, Jira).

Constructive Quality Management

Many of the functions of a software system are not created directly by the development team, but are instead reused by deploying pre-existing program code, known as libraries or frameworks. Depending on the maturity and distribution of a programming language, there are often mature frameworks for standard tasks (such as database connections), as well as for special functions (such as visualization).The permitted frameworks are often specified for a system’s central functions so that the degree of heterogeneity within the entire application landscape is kept low.

Interpersonal Measures

**Interpersonal measures** are activities that aim for a professional and positive cooperation between the individuals involved in the process. These measures are typically expressed through qualiﬁcations, joint activities, and the design of the working environment. Qualiﬁcations include the development of personal and professional skills through training or coaching. Joint activities are activities that the project team carries out together, but are not directly related to the project objective. Examples of this are the joint implementation of professional training, regular joint leisure activities (such as sports or social engagements), or team events (such as barbecues or outdoor events). The design of the work environment is also a constructive measure. For example, creating designated quiet zones can enable concentrated work, putting team members together in project offices can promote communication within the team, and using kanban boards can support self-organized work.

###### Examples of Constructive Quality Management

The following examples illustrate the many possibilities of specific constructive QA measures:

* Application of a process model that explicitly provides for risk assessment in the development process at every stage.
* Using a tool for conﬁguration management of source and object code.
* Sample document for a quality management plan with a predefined outline and instructions for completion that ensures that all important points are covered.
* Establishing programming guidelines that require, among other things, the naming of variables and making unit tests mandatory for every business function in the program code.

Interpersonal measures aim for positive cooperation between the people involved in the process.

Examples of this are joint activities or an appropriately designed work environment.

### Selected Techniques

###### Root Cause Analysis with the 5–Why Method

Within the scope of a root cause analysis, an attempt is made to identify the actual error causing error symptoms. As a pragmatic technique for error analysis, Kleiner (2013, p. 229) recommends the *5–Why Method* as presented by Ohno (2006). The core idea of this analysis technique can be summarized with the following sentence: “Ask ‘why’ ﬁve times about every matter”. The question “why” should be asked five times in a row about an error, then it is very likely that the original cause will have been found.

The following example illustrates the application using a typical misbehavior:

“A higher than average number of errors are reported for the shopping cart component.”

1. Why question: Why are errors reported? Answer: Because the calculated total amount often deviates somewhat from the total of the individual items.
2. Why question: Why does the total amount differ? Answer: Because rounding errors occur when calculating discounts for premium customers.
3. Why question: Why do rounding errors occur? Answer: Because they are not stored with the required accuracy during the internal calculation of intermediate results.
4. Why question: Why can't the accuracy be achieved? Answer: Because a data type that only saves up to the second decimal place was chosen for storage.
5. Why question: Why was this data type chosen? Answer: Because the developer always assumed that no more than 2 decimal places would need to be stored for the amounts.

In this case, the original cause of the error can be localized with five consecutive why questions. To eliminate the error in the long term, targeted training measures can be carried out and/or a checklist for the code review can be expanded to include an examination for the risk of rounding errors. When using the 5–Why method, a root cause analysis involves the following steps: (cf. Kleiner 2013, p. 229):

Step 1: Collection of all available information about the error and the erroneous behavior triggered by the error.

Step 2: Application of the 5–Why method.

Step 3: Identiﬁcation of the places in the program code that must be changed to eliminate the error.

Step 4: Identiﬁcation of possible QA measures (constructive and analytical) to prevent the error in the future.

Constructive Quality Management

Step 5: Introduction of the QA measures.

Step 6: Evaluation of the QA measures to determine whether they reliably prevented the errors.

###### Timeboxing

Timeboxing refers to a project planning technique that provides a fixed and unchanging time frame for activities and procedures within a project. This technique is widely used in agile software development and is frequently deployed as a tool for efﬁciently conducting meetings. Timeboxing helps prevent lengthy and sprawling discussions and leads to high meeting quality by setting a strict framework. The core idea of timeboxing is to allocate time to the individual activities or agenda items and to consistently monitor and adhere to the allotted time. If an activity or agenda item cannot be completed in the allotted time, it is terminated at the stage reached and the next activity or agenda item is started. A fixed time frame is usually set for meetings, which is made known to all participants. The meeting ends after the time has expired, regardless of its progression.

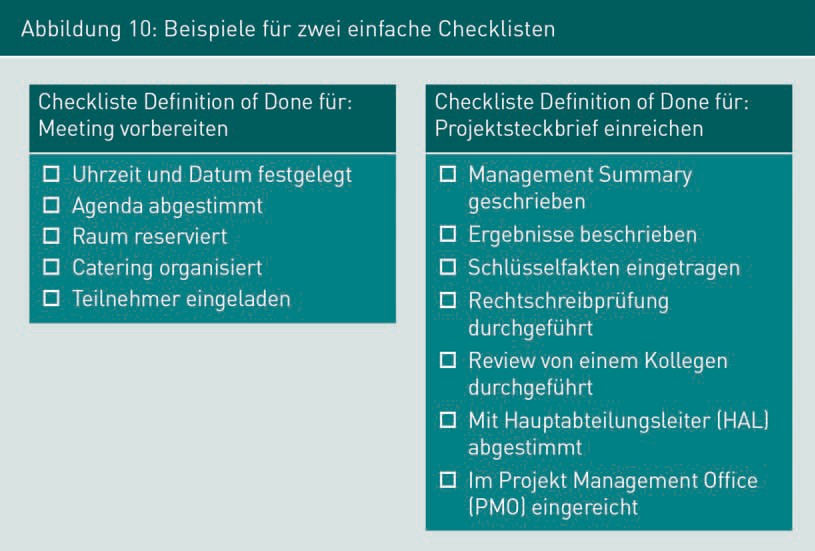
When timeboxing is newly introduced, a moderator must be designated to monitor adherence to the schedule. They must have the right to end an agenda item or the meeting. In practice, observations show that after a short period of familiarization, the time limit introduced by timeboxing becomes internalized by the participants, meaning that the need for a moderator decreases over time. For this purpose, consistent adherence to the time limit is necessary, particularly in the introductory phase, in order to make it clear that not all planned points can be discussed and that discussions can be broken off.

###### Checklists

Checklists are an important tool in many different areas of everyday life, and the same applies to industrial processes. They help in the packing of private luggage before long journeys or mountain expeditions, and assist airplane pilots in conducting a system check before taking off in an airplane.

They are used to ensure that no required activities are forgotten during a task and/or that all the required result or management artifacts are available at the end of the task. Checklists are usually created for recurring standard tasks, such as training new employees, setting up a development environment for a project, and examining the current process step at quality gates.

Figure 10 below shows two simple checklists from practice: one on the left for preparing meetings and one on the right for submitting a project profile on the basis of which the project will be approved or rejected.



As shown in Figure 10 above, checklists consist of a simple list of entries that can be easily checked off after review or completion. The following points should be considered when formulating checklists:

* + One item on the checklist corresponds to exactly one objective; compound objectives should be avoided and divided among several items.
  + Clear and simple wording of the checkpoints increases readability and understandability. A checklist should not be larger than an A4 sheet of paper so it can be easily printed out with any printer and all the checkpoints can be understood at a glance.
  + The criteria required to fulfill a checkpoint must be transparent; if necessary, sub-checkpoints or separate checklists are created for individual checkpoints.

Checklists must also be subject to continuous monitoring. If necessary, they must be adapted in order to react to changed framework conditions or quality requirements.

Summary

Constructive quality management measures serve to ensure the quality of the design process. They can be differentiated according to organizational, technical, and interpersonal measures. In general, it is not sufficient to simply define measures for constructive quality management. Compliance with them must be monitored and explicitly enforced during the course of the software process.

Constructive Quality Management

In addition, the effectiveness and quality of the specifications must be examined regularly and the specifications adjusted, if necessary. Specific examples of measures in constructive quality management are the specification of root cause analysis in order to identify the causes of errors, as well as timeboxing, which can be used to organize and conduct meetings, and standardized checklists, which can be used to ensure process quality.



# Unit 4

## Static Quality Assurance: Assessment and Measurement

#### STUDY GOALS

After completing this unit, students will know ...

... where static procedures are used for quality assurance.

... what the roles and activities in the different review techniques are.

... how metrics are used for software systems and what the typical software metrics are.

... what static code analysis is and what it is used for.

DL-D-IQSS01–L04

1. Static Quality Assurance: Assessment and Measurement

### Introduction

In contrast to constructive quality management, which specifies activities in a software process, static quality assurance measures examine and evaluate the results created in a software process. Following an overview of static quality assurance procedures, this unit introduces and discusses various review techniques as well as the use of metrics. In addition, examples of uses of static code analysis techniques are provided.

### Use and Overview of Static Procedures

Review A review technique is a static manual test procedure for analyzing a test object. The test object is analyzed, assessed, and investigated. The information obtained in the process is compiled, summarized, and

evaluated.

Static Code Analysis This is the tool-supported content analysis and evaluation of a program code, but without executing it.

Static quality assurance procedures are analytical procedures in which the test object is analyzed, assessed, and investigated within the scope of a **review**. The information obtained in the process is compiled, condensed into metrics or key figures, if necessary, and, ultimately, evaluated. As already shown in Figure 1 above, in contrast to dynamic procedures, the program code to be examined is not executed in static procedures. In principle, all static quality assurance procedures can be conducted without the use of software, i.e., only “with pen and paper” as it were. However, the measurement and calculation of metrics, as well as **static code analysis**, are particularly well supported by tools. Static procedures can be used to examine all artifacts created in the software process, as well as activities for quality assurance. In addition, organizational specifications such as the software process model or the quality plan can be examined as part of quality control activities.

Figure 7 above shows an example of a software process with quality management activities. Static procedures can be used within the scope of the following activities:

\*1 = Examination of the business requirements: review techniques are used to examine documented requirements with regard to predefined quality criteria.

\*2 = Quality gates: with the help of checklists, the current situation in a software process is assessed and compliance with required attributes is evaluated.

\*3 = Examination of the technical speciﬁcation: review techniques are used to examine the technical speciﬁcation with regard to predefined quality criteria. Technical data schemas and data structures are also automatically analyzed and evaluated, if required.

Static Quality Assurance: Assessment and Measurement

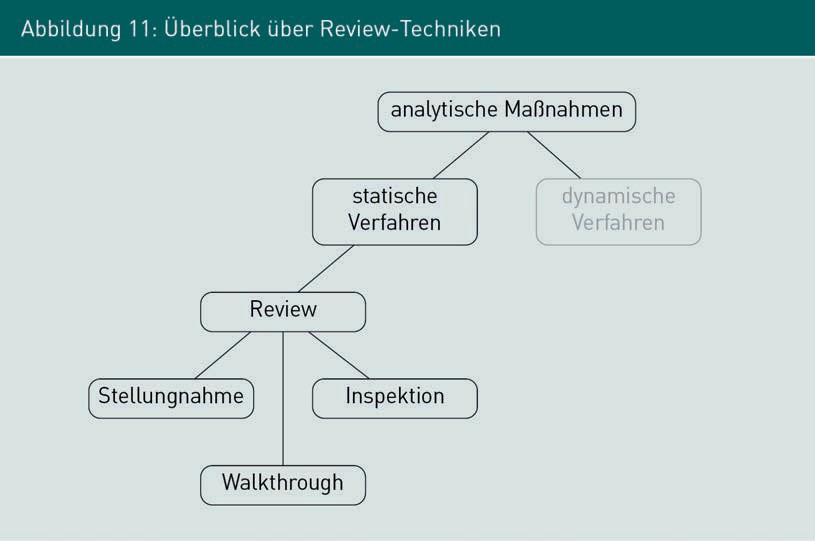
\*4 = Evaluation of the planned architecture: review techniques and scenario-based architecture analysis are used to examine the suitability of the architecture description for meeting the requirements. Detailed technical designs of components and data structures can be automatically analyzed and evaluated, if required.

\*5 = Test case generation: the test cases generated are reviewed using review techniques to determine whether they can ensure the quality level defined for the current project.

\*6 = Component tests: although component tests are not static procedures, measures for static code analysis are carried out in the context of these tests. The program code created is automatically reviewed for compliance with required attributes and error probability analyses are carried out.

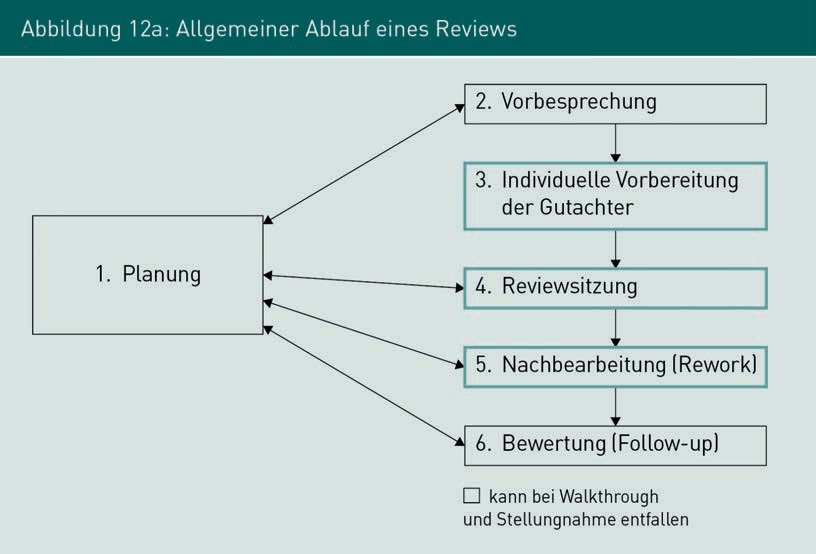
### Assessing with Review Techniques

A review technique is a static manual test procedure for analyzing a test object. Figure 11 below shows an overview of typical review techniques, which differ in terms of effort and expense, as well as degree of structuring. For example, an opinion is a very simple and not very elaborate review technique, whereas an inspection is a very highly structured and elaborate review technique.



Walkthrough This is a review with multiple reviewers, but without adherence to formal criteria.

Figure 12a below shows the general review process. Depending on the type, objective, and organization of a review, individual phases may be carried out less intensively or not at all. An inspection is a very formal review that includes all the activities shown in Figure 12a. In a **walkthrough**, only the activities shown with a thick border in Figure 12a are often carried out, but they are formally organized. The formal organization can be omitted in an opinion, which is therefore the simplest review technique. A detailed description of the individual activities is found beneath the description of the roles.



###### Roles in Review

When conducting reviews, the roles of moderator, reviewer, author, and recorder are involved depending on the selected review form. The specific use of these roles varies depending on the review variant selected, as shown in Table 3 below.

The moderator is responsible for planning, preparing, and leading the review session. They select the other persons involved in the review, often in collaboration with the author. In addition, they are responsible for monitoring the correction of deficiencies. The reviewers are experts who, based on their knowledge and experience, are able to evaluate the functional quality of the test objects and, if necessary, identify deficiencies. Depending on the complexity and criticality of the test objects, several reviewers can also be involved in the review, with typically one to five reviewers, or more if necessary. However, the effort and expense required for coordination and harmonization also increases with each additional reviewer. The author is available to the reviewers and the moderator to answer questions about the test object and provide clarification in case of ambiguity.

Static Quality Assurance: Assessment and Measurement

They are responsible for eliminating deficiencies in the test object during the course of a rework. The recorder is responsible for preparing the review minutes during the review session.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 3: Roles Involved in Review Techniques | | | |
| Role Name | Inspection | Walkthrough | Opinion |
| Moderator | X | X  (handled by the author, if necessary) | - |
| Reviewer | X | X | X |
| Author (of the test object) | X | X | X |
| Recorder | X | – | – |

###### Review Activities

The activities of a review shown in Table 4 below are described in detail.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 4: Review Activities | | | |
| Activity | Objectives | Result | Person Responsible |
| Planning | * Organization of all resources required (such as people, appointments, rooms). * Determination of the review objectives. * Determination of the existing criteria. * Designation of the reviewer. | * Scheduling coordinated with all parties involved. * Documented test objectives and pass criteria. | Moderator |
| Preliminary discussion, if required | * All parties involved receive an overview of the test object and the review objectives. * If applicable, breakdown of the review aspects/views for the reviewers. * Distribution of all required documents. * Clarification of organizational and functional issues. | * Each reviewer knows the test object, the review objective, and the review criteria for which they are responsible. | Moderator |
| Preparation by the individual reviewers | * The test object is worked through individually by the reviewer and examined using the specified criteria. | * Documented questions by the reviewer, as well as documented content-related deficiencies, and formal deficiencies (such as spelling mistakes). | Reviewer |

Static Quality Assurance: Assessment and Measurement

|  |  |  |  |
| --- | --- | --- | --- |
| Activity | Objectives | Result | Responsible  Person |
| Review session | * Joint walkthrough of the test objects by all parties involved. * Discussion and evaluation of the functional deficiencies identified by the reviewer. | * Log of all functional deficiencies identified. * Collection of all formal errors (not discussed in session). * Documented result (no deficiencies, rework necessary, termination due to serious deficiencies). | Moderator |
| Rework | * Correcting the deficiencies identified. * For inspection: prepare a deficiency report and a list of corrections made. | * Reworked test object. * Deficiency report, to-do list. | Author |
| Evaluation | * Examine whether all corrections have been implemented. * Final evaluation of the test object | * Review report | Moderator |

###### Comparison: Opinion, Walkthrough, Inspection

The opinion, walkthrough, and inspection review techniques are described as follows.

Opinion

The simplest and quickest review technique is an informal opinion. In this process, the test object is read by a person who is uninvolved in and independent of its creation, and evaluated with regard to the relevant review criteria. Deficiencies are identified and briefly

justified. For example, the opinion can be given by a colleague or an external service provider. The process does not follow any particular pattern or specifications. The following scenario outlines an example of an opinion: The requirements document is handed over to a test engineer. The test engineer examines the document to review the testability of the requirements (Figure 12, 3.), identifies the deficiencies, and gives the requirements engineer feedback on improvements (Figure 12, 4.). The requirements engineer then makes improvements (Figure 12, 5.).

Walkthrough

In contrast to an opinion, a walkthrough involves the document being read by several people, with the identiﬁed deficiencies being discussed and evaluated within the group. Like an opinion, a walkthrough does not need to meet any formal criteria, but the effort and expense required for organization and coordination is greater due to the participation and discussion of several reviewers. Depending on the needs and the project situation, the role of the moderator can be conducted by the author or by a neutral person. By using multiple reviewers, a more detailed review can be carried out with a walkthrough than with an opinion. For example, perspective-based reading can be used to specifically consider different aspects during a walkthrough. Furthermore, by involving multiple people as reviewers, a common, intra-team understanding of the content of the test objects can be obtained and ensured. For example, if each developer participates in the speciﬁcation review as a reviewer, a uniform level of knowledge within the team during the review session is achieved in the joint discussion with the author regarding the content and possible deficiencies.

Inspection

An inspection is a formal review that generally involves several reviewers and is carried out according to a defined procedure and whose execution and results must be documented in detail. The organizational and implementation complexity is greater compared to an opinion or walkthrough, since the roles of a moderator and recorder are explicitly required in an inspection. An inspection is conducted on important test objects where unidentiﬁed deficiencies have a very high potential for damage. Typical scenarios are milestones in very large projects and/or artifacts in highly distributed development projects involving multiple organizations. With a focus on documentation and formal frameworks, inspections are also used when guidelines or legislation require demonstrable risk mitigation measures. A comparison of the review techniques in terms of characterization and use is shown in Table 5 below.

Static Quality Assurance: Assessment and Measurement

|  |  |  |  |
| --- | --- | --- | --- |
| Table 5: Comparison of Review Techniques | | | |
|  | Opinion | Walkthrough | Inspection |
| Characteristic | * Quick, informal review without explicit organization. | * Informal review by several reviewers. | * Formal review by several reviewers in line with a deﬁned scheme. |
| Application scenarios | * Receive quick, informal feedback on the test object. * Internal feed- back among colleagues without an obligation to prove formal review criteria. | * Detailed examination of important artifacts in which unidentiﬁed errors can cause clearly appreciable damage. * Creation of a uniform understanding of facts and promotion of team communication. | * Examination of important artifacts in which unidentiﬁed errors have a very high potential for damage. * Obligation to prove the formal criteria of the review. |

###### Feedback Technique: Feedback Burger

It is often difficult to communicate feedback on a work result, particularly when errors or deficiencies have been identified in a review. Communicating criticism together with the motivation to subsequently revise the result is often a challenge, especially since, for example, in requirements engineering, maintaining the stakeholders’ willingness to cooperate is an important success factor. The **feedback burger** technique supports the review process by combining different principles for communicating feedback in a clear manner.

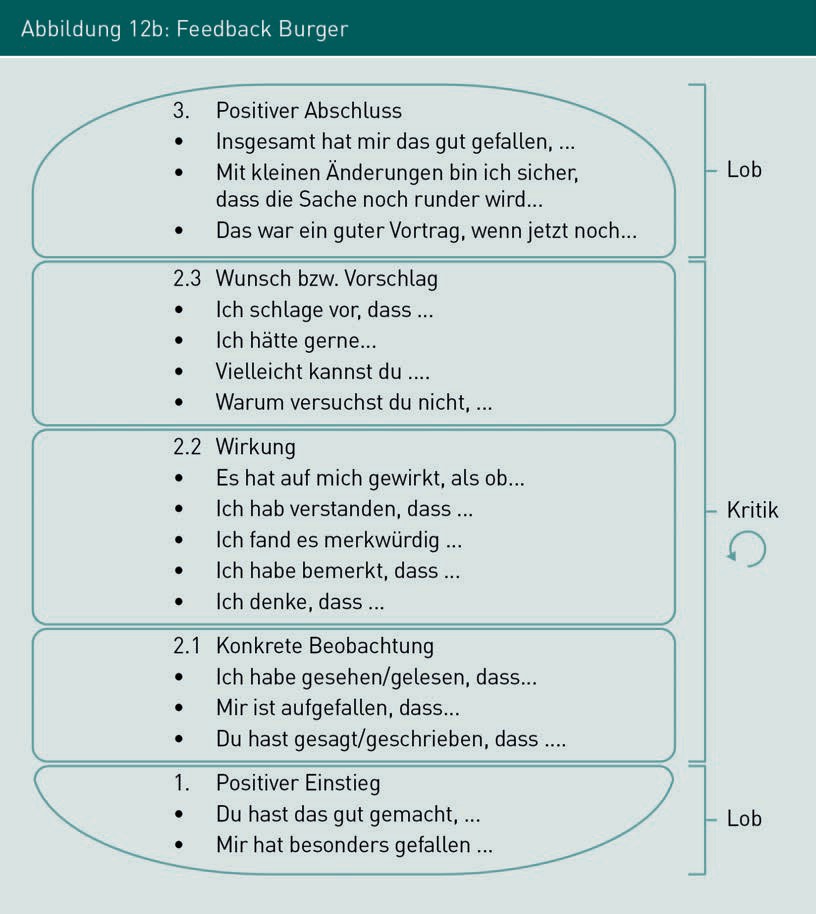
The following steps are important when preparing to communicate feedback:

Feedback Burger This is a technique for communicating feedback on work results and serves to maintain the willingness of the stakeholders involved to cooperate.

1. First, read (and/or listen) with as little bias as possible: the goal here is to develop an understanding of what is written/said and to evaluate it.
2. Prepare feedback: the content of the review is evaluated with the help of an evaluation scheme, for example, using the test criteria previously defined. All assertions in the feedback should always refer to specific observations/positions within the review. The feedback burger helps prepare the feedback. The objective of a review is not to criticize the author personally, but to contribute to the improvement of the result.
3. Giving feedback (in a review session): when giving feedback, ensure that it is clearly shown as a very personal impression (“I” comments) and avoid generalizations such as *always*, *never* or *everything*.

Figure 12b shows the scheme of a feedback burger that can be used to prepare and communicate feedback. The basic structure of the feedback burger is a combination of communicating errors and deficiencies as well as positively perceived characteristics. The burger is built from the bottom up. The feedback should start with a positive introduction, in which specific positive details and impressions are mentioned. Possible criticism or deficiencies are then presented on the basis of specific, individual observations. This is done with the pattern *observation-perception-wish*: first, the specific observation is mentioned, then the individual effect on the reviewer, concluding with specific requests and suggestions for improvement. This cycle can be repeated several times. The conclusion of a piece of feedback according to the feedback burger is a short, positive summary that motivates the author to consider the suggestions for change in the rework.

Static Quality Assurance: Assessment and Measurement



### Measurement and Metrics

Measurement is another technique for determining the quality of products and processes with static procedures. In principle, the attributes of artifacts created in the software process as well as the attributes of the software process itself can be measured. The function that determines a numerical value from the results of measuring a test object is called a metric.

Software Metrics Example:

The lines of code (LOC) metric can be determined for a software system. For this purpose, the number of all programmed lines in the program code is determined and specified in numeric form.

###### Use of Metrics

The objective to be achieved with the use of metrics is the quality control and improvement of results and processes. Following the guiding principles of “To measure is to know” as well as “You can’t control what you can’t measure!”, the values of parameters are determined and analyzed with the help of tools. Assertions regarding the current state of the product or process are derived based on the measured values. Fundamentally, metrics for the evaluation of processes can be distinguished from metrics for the evaluation of artifacts. Examples of aspects to be measured in a software process are the resources consumed (time, money, personnel), number of errors identified in the project, duration of individual activities, mean correction time of identiﬁed errors, or the number of changes to a system component. Aspects that are measured for results are, for example, scope (number of pages, LOC, number of model elements), complexity, quality (number of identiﬁed functional errors, number of identiﬁed formal errors), stability (extent of changes in a given period of time), or style (adherence to conventions, readability, freedom from redundancy).

###### Examples of Software Metrics

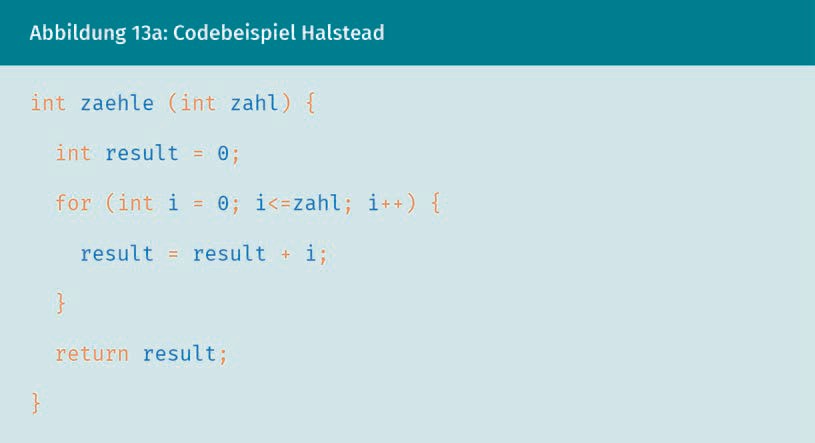
In addition to relatively simple metrics such as LOC, there are also more complex metrics whose value is determined from several measured values.

Textual Complexity with Halstead Metrics

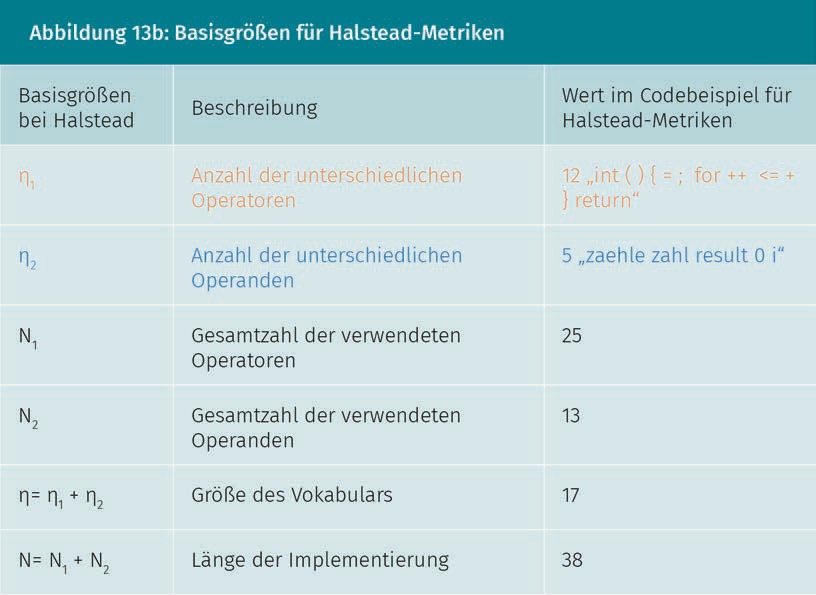
Halstead metrics (see Liggesmeyer 2009, pp. 260–262) were developed to determine the complexity of software components. First, specific basic values are measured in the program code from which various metrics are then calculated.

The following code example is used to explain Halstead metrics as an example of software metrics:

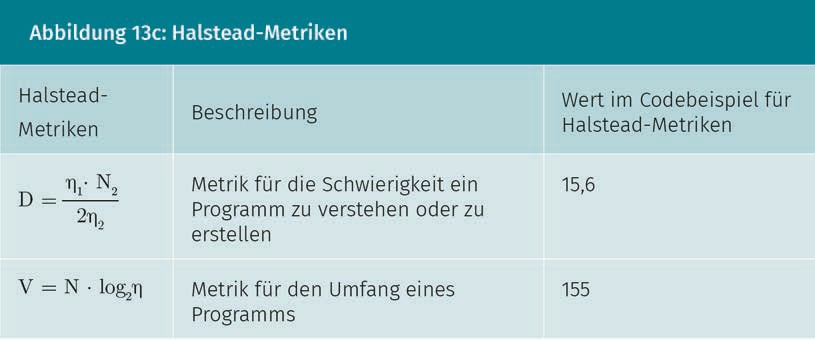
Static Quality Assurance: Assessment and Measurement



For this code example, the basic quantities of the Halstead metrics are determined in Figure 13b below, the basis of which is the determination of the type and number of operators and operands in the program code. Note: how to specifically count differs depending on the programming language and implementation of the metric.



Metrics with further complexity can be determined by drawing on the basic variables. Examples for further Halstead metrics are provided in Figure 13c below

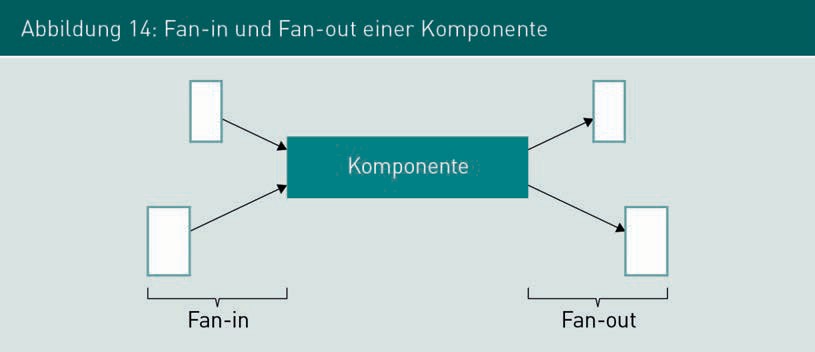


The Halstead metrics were developed in 1977. Nowadays, there are many different metrics, each of which refer to very specific aspects of a program code. The Halstead metrics are examples of the measurement of textual complexity and can be applied to any programming language.

Structural Complexity Metrics

The fan-in and fan-out metrics illustrated in Figure 14 are frequently determined to measure the structural complexity of components:

* Fan-in: metric for a component that determines the number of functions outside a component that access the functions of the component under consideration;
* Fan-out: metric for a component that returns the number of functions accessed from the component under consideration to other components.



Static Quality Assurance: Assessment and Measurement

Metrics for Object-Oriented Systems

For examples of typical metrics for object-oriented systems, see Grechenig (2010, pp. 476–479):

* + Depth of inheritance trees, i.e., length of the maximum path of an inheritance tree.
  + Depth of a class (DIT), i.e., the length of the path from the root to a class in the inheritance tree.
  + Number of immediate subclasses of a class (NOC).
  + Coupling between objects (CBO): number of classes that use methods of the measured class. The higher the coupling between objects, the more complex it is to test or reuse the classes.
  + Response for a class (RFC): number of all possible executable methods of a class, including both the methods that are implemented in the class as well as the methods that can be accessed (by linkage) in other classes.

###### Metrics Discussion

Metrics are key performance indicator systems and thus have their advantages and disadvantages.

Advantages

Software metrics are relatively easy to determine. Plug-ins, with which the values can be determined and displayed at the push of a button, are available for common development environments. Similarly, metrics for a software process can be easily determined, provided that the underlying database is continuously updated. Furthermore, most metrics are independent of a specific programming language or a project organization and can therefore be easily used in most projects. Thus, software metrics, in particular, offer a simple means of determining a wide range of attributes of result artifacts.

Disadvantages

Metrics can only be used to determine measurable attributes of systems or processes. However, it is often not possible to make assertions about the actual consequence of certain attributes or combinations of attributes, particularly with software metrics. Studies show, for example, that the variables measured using Halstead metrics correlate with the actual reading and comprehension time. However, it is not generally possible to derive reliable actual consequences on the basis of software measurements. The measurement result that the system is easy to understand and its structural complexity is not very high does not say anything about whether the system meets the customer’s requirements.

If metrics are also included as components in objective agreements, there is a risk that the fulfillment of the metric will become more important than the fulfillment of the business requirements. The following example illustrates this risk: developers are to be evaluated on the basis of the LOC they have created. The lines of code metric, which measures the instructions as well as the source code comments, creates an incentive to write as many comments as possible.

In contrast, the non-commented source code metric, which only measures the number of lines of code without source code comments, creates an incentive to do without comments and write as little program code as possible in a line.

Metrics only have very limited suitability for deriving specific assertions about the quality of systems or processes. Nevertheless, they can be used as an indication of possible sources of error. For example, during a manual code review, the focus can be placed on particularly large code sections or code sections that have been evaluated as complex.

### Static Code Analysis

Static code analysis procedures are used for the qualitative evaluation of program code. In contrast to metrics, static code analysis does not measure any attributes of the program code, but rather analyzes and evaluates the code content. Typical areas of application for static code analysis are automatic style checking and analysis of typical error patterns. A number of tools for static code analysis are available for industrially distributed programming languages such as Java or C++ and can be integrated both as a plug-in in development environments and in automatic build processes, thereby ensuring that the analysis is performed automatically and repeatedly. In the following, three different tools for the Java programming language are presented as examples, all of which are freely available.

###### Automatic Style Analysis with Checkstyle

The Checkstyle program is a tool for automatic style analysis that checks program code for compliance with programming conventions. Examples of this are the correct use of annotations, compliance with naming conventions for classes, attributes, methods, and variables, and checking for the presence of Javadoc comments. A complete overview of available checks is documented on the Checkstyle website (n.d.). The specific conventions to be checked can be conﬁgured in detail, so that in addition to the standard conventions for Javacode, custom conventions can also be created and then automatically checked.

However, mere adherence to programming conventions does not lead to an error-free system in itself. Particularly in the case of manual code reviews or development in a team, in which several developers are working on one component, automatic style analysis can be used to promote familiarization and code comprehension. In addition, uniformly formatted program code makes it easier to work with version control systems. With consistent formatting, fewer formatting changes, such as line indentations, are displayed in change tracking.

Static Quality Assurance: Assessment and Measurement

###### Source Code Analysis with PMD

Like Checkstyle, the PMD program analyzes the source code of programs. Here, however, the search for possible error sources is the focus. Typical examples of these kinds of code sequences that can be found with PMD are:

* + possible bugs due to empty try/catch or switch blocks,
  + variables, parameters, and private methods that have been created but not used,
  + excessive use of String and StringBuffer methods,
  + identiﬁcation of unnecessary or overly complicated if-assertions, as well as
  + code sequences duplicated using copy & paste.

An overview of all the rules supported by PMD is available on the Internet (PMD n.d.).

###### Bytecode Analysis with SpotBugs

The SpotBugs program is a tool for analyzing Java bytecode for typical error patterns. The source code is not required for this, but the program must compile successfully for analysis with SpotBugs.

A current overview of the available error patterns can be accessed on the Internet (SpotBugs n.d.). SpotBugs divides the error patterns into categories. Examples of these are:

* + bad practice: for example, implementing the equals() method without the hashCode() method, or string comparisons with == instead of equals(),
  + correctness: for example, the incorrect use of the increment function (i++) in return assertions or adding a collection object to itself, and
  + dodgy code: for example, storing a value in a local variable that is never read,

as well as the categories of malicious code vulnerability, multithreaded correctness, and performance.

Both source code analysis with PMD and bytecode analysis with SpotBugs are tools for identifying actual errors. Each analysis tool provides indications of where “good practice” is violated in the code. However, whether these are actual errors must be manually examined by the developers.

With static code analysis, however, typical errors can be identiﬁed at the programming phase, meaning that the probability of typical volatility errors can be reduced during subsequent tests. Nevertheless, there is no guarantee that program code without any error patterns will actually work.

Summary

Static quality assurance procedures are analytical procedures in which the test object is analyzed, assessed and investigated within the scope of a review and the information obtained is compiled and condensed into metrics or key figures, if necessary, and then evaluated. In contrast to dynamic procedures, the test object is not executed here.

Review techniques are manual static review procedures that differ in terms of effort and expense, as well as degree of structuring. For example, an opinion is a very simple and less complex technique, whereas an inspection is a very highly structured and elaborate review technique. A walkthrough is less structured than an inspection, but more structured than an opinion.

Another technique for determining the quality of products and processes with static procedures is the determination of software metrics by measuring program code attributes of the system. Examples of this are Halstead metrics for measuring textual complexity.

In addition, procedures for static code analysis can be used for the qualitative evaluation of program code by analyzing and evaluating the content of the code. Typical areas of application for static code analysis are automatic style checking and analysis of typical error patterns in the code.



# Unit 5

## Dynamic Quality Assurance: Testing

#### STUDY GOALS

After completing this unit, students will know ...

... what a test case is, which test levels are distinguished between, and what the typical techniques for test case generation are.

... what use-case-based test case generation is and how it is used.

... how to generate test cases with equivalence partitioning.

... how test cases are created with state-based test case generation.

... how the generation of random data can be used.

DL-D-IQSS01–L05

1. Dynamic Quality Assurance: Testing

### Introduction

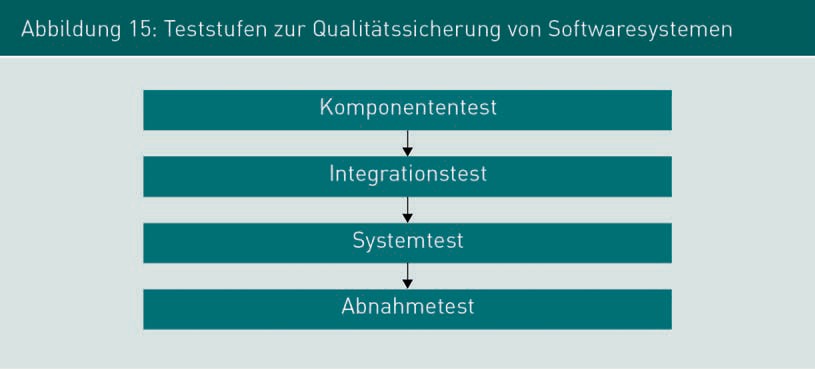
In contrast to the static procedures of analytical quality assurance, the term *testing* refers to conducting dynamic procedures. The test object in dynamic quality assurance is always a piece of software. In testing, the software is executed with specific input values and the outcome of its execution is evaluated.

### Use and Overview of Dynamic Procedures

Dynamic procedures are a form of analytical QA measures. They are also referred to as testing procedures, software tests, or, simply, tests. When testing software, a system or parts of a system are executed, specific input values are transferred, and the system behavior or the generated result is evaluated. The tests are therefore diagnostic measures for examining and evaluating the quality of software artifacts after they have been created.

###### Test levels

Testing activities do not occur after all development activities have been completed, but rather during development. Therefore, in industrial software development, testing is started as soon as the first software artifacts have been created. Figure 15 below shows a schematic overview of the different test levels in software engineering.



The specific quality attributes that are the focus of the current test are not only determined by the quality objectives defined for the project, but also by the current test level. Therefore, the specific test cases during testing differ depending on the test level.

Dynamic Quality Assurance: Testing

###### Test Case

A test case serves as a guideline for the execution of software tests and consists of the following elements, as a minimum:

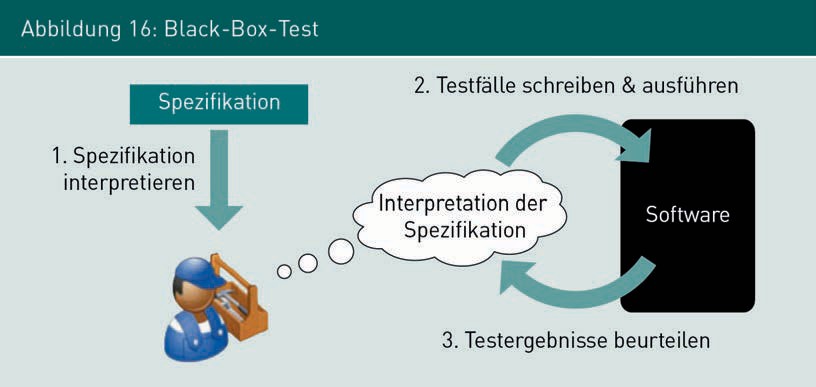
* Preconditions that must be ensured before the test step.
* Test data or test actions that are entered or carried out during the test step.
* Post-conditions, the fulfillment of which is examined after the test step.
* Descriptive data regarding the test case, such as name, ID, relevant component, function, or attribute being examined, use case, or name of the creator.

###### Test Case Generation

The literature on software testing distinguishes between different test techniques on the basis of how the test cases are generated. A distinction can be made between black box tests and white box tests.

Black Box Testing

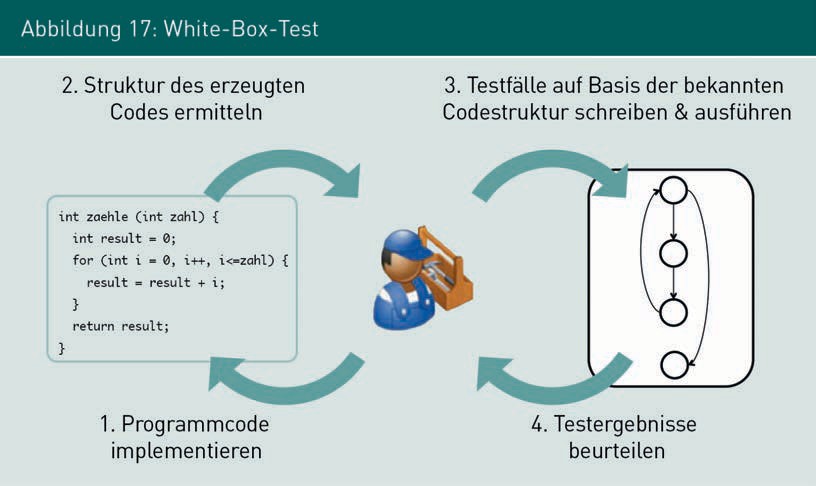
The objective of black box testing is to examine all the functions and attributes required in the speciﬁcation for a system or system component. When generating a test case, the focus is on covering as many of the speciﬁed functions of a system as possible. No information is available about the internal structure of a test because only the externally visible behavior is relevant. Figure 16 below schematically illustrates the procedure for black box testing: the tester reads and interprets the speciﬁcation. Then, based on their interpretation of the speciﬁcation, they generate the test cases and execute them. The tester finally evaluates the test results based on their interpretation of the speciﬁcation.



Black box tests are generally used to examine the interaction between various system components. They are very user-oriented and are typically used for integration, system, and acceptance tests. The speciﬁcation takes on central importance for black box tests, in particular: the degree of test coverage is determined on the basis of the speciﬁcation, it is used as the basis for generating the test cases, and the test results are evaluated on the basis of the speciﬁcation. It is therefore all the more important to ensure that all the functions and attributes actually required by the stakeholders are documented in the speciﬁcation.

White Box Testing

White box tests are generated with knowledge of the program code, which, in contrast to black box tests, is known in this instance. The objective of white box testing is to cover as many of the assertions and control structures (such as if, while, for, switch) implemented in the program code as possible in order to test as many of the implemented instructions as possible. Figure 17 below schematically illustrates the procedure for white box testing. After a part of the program code has been implemented, the structure of the code that has actually been generated is analyzed. On this basis, test cases are then formulated to achieve the highest possible coverage of the program code created. After execution, the results are evaluated and, if necessary, the program code is modified before being tested again. The roles of developer and tester are often assumed by one person, which is why these tests are also called developer tests.



In general, white box tests accompany implementation and are carried out for component and integration tests.

In addition, Liggesmeyer (2002, pp. 50–51) also distinguishes between function-oriented and control flow-oriented tests.

Dynamic Quality Assurance: Testing

###### Test Coverage

In practice, it is fundamentally not possible to test industrial information systems fully and continuously. The objective of QA measures is therefore not complete quality assurance, but rather cost-optimized quality assurance. When conducting tests, testers must limit themselves to a targeted selection of test cases and test data. One measure of the completeness of test cases is known as **test coverage**. Depending on the method applied, test case generation can be used to determine how “completely” the set of formulated test cases examines the system. In practice, test coverage is used to limit the test activities required for the project.

###### Test Case Selection

In both black box and white box testing, a system or part of a system is executed with specific input data. Since not all theoretically possible combinations of input data can be tested in practice, testing information systems always follows a sampling procedure. The correctness of systems cannot be proven with tests. Therefore, one of the main challenges in testing is the selection of suitable test cases and test data so that the highest possible test coverage can be achieved with the least possible effort and expense. In short, from the population of all possible input values, exactly those with which as many potential errors as possible can be identiﬁed must be selected.

Table 8 below presents and describes different techniques for selecting test data, along with their characteristic attributes.

Test coverage

This is the measure of completeness of test cases. The specific measured value depends on the type of test and the selected technique of test case generation.

|  |  |  |
| --- | --- | --- |
| Table 8: Techniques for Test Data Determination | | |
| Technique Name | Description | Operation |
| Use case-based test case generation | Derivation of test data from documented use cases, for which functional preconditions, the objective to be achieved, and post-conditions are often documented. | Testing whether the defined business objectives can be achieved within the use case; usually only applicable as a black box test. |

|  |  |  |
| --- | --- | --- |
| Technique Name | Description | Operation |
| Equivalence partitioning | Grouping of possible input values with which the same functional result is achieved in equivalence classes. Each equivalence class is tested by a representative. | Determination of input data for specialist functions, targeted testing of system behavior with valid and invalid values. |
| Boundary value analysis | Generation of test data targeted at the boundaries of valid and invalid value ranges, often in combination with equivalence partitioning. | Testing the system behavior specifically at valid and invalid values or at the limits of equivalence classes, each of which requires a different functional or technical behavior. |
| State-based test case generation | Generation of function test data based on the life cycle or functional states of business objects (black box testing); generation of technical test data to review the technical states (white box testing) of a system, component, or class as well as their state transitions. | Functional or technical state testing that can also be used to explicitly test the non-execution of forbidden state transitions; particularly relevant if state diagrams or state tables are used in the speciﬁcation of system behavior. |
| Generation of random test data | Automatic, non-determi- nistic generation of test data; data are usually not generated completely randomly; it is often necessary for the structure of input data to meet specific requirements. | Supplementation of structured methods for test data generation; suitable, for example, in combination with equivalence partitioning. |

Dynamic Quality Assurance: Testing

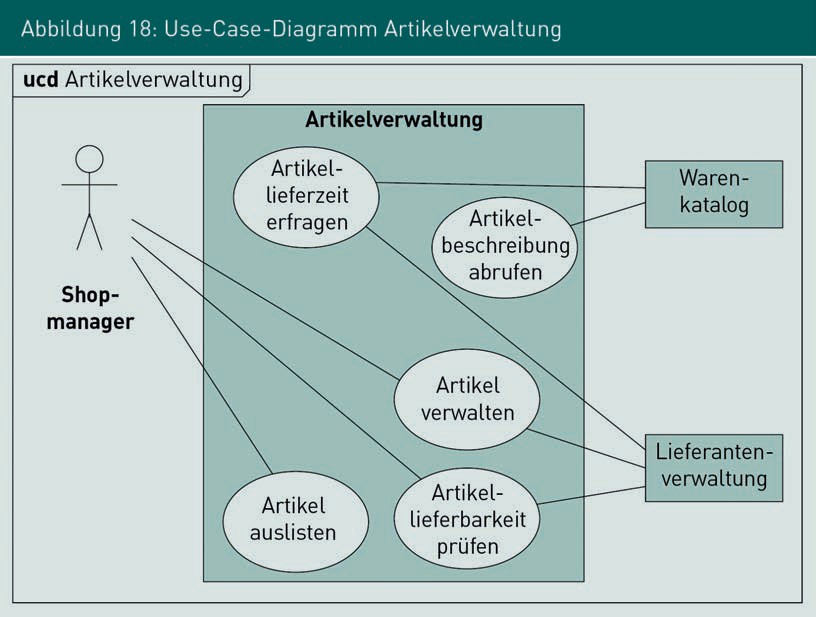
|  |  |  |
| --- | --- | --- |
| Technique Name | Description | Operation |
| Cause and effect analysis  (see Liggesmeyer 2002, pp. 66–76) | Testing of relationships and interactions between equivalence classes; the outcome of this testing is combinations of different equivalence classes, each resulting in an entirely determined system behavior. | Testing of application behavior (effect) that the combination of different input values determines (causes). |
| Instruction coverage  (see Liggesmeyer 2002, pp. 85–88) | Generation of test data with which each instruction in the code (white box test) is executed at least once; generation of test data with which each business function (black box test) of the supported business process is executed at least once. | Ensuring that each relevant instruction/function is accessed once during the test. |
| Branch coverage  (see Liggesmeyer 2002, pp. 88–93) | Generation of test data with which each branch in the control ﬂow is run through at least once; i.e., decisions in the control flow must be made in such a way that each outgoing branch is run through at least once. | Ensuring that every possible branch in the source code (white box test) or every possible branch in the business process (black box test) can be reached and run through. |
| Condition coverage  (see Liggesmeyer 2002, pp. 93–117) | Generation of test data that allows each individual logical decision in the program code (white box test) or at decision nodes in the business process (black box test) to be evaluated at least once as TRUE and at least once as FALSE. | Testing of composite decisions and the system behavior resulting from their behavior. |

|  |  |  |
| --- | --- | --- |
| Technique Name | Description | Operation |
| Path testing (see Liggesmeyer 2002,  pp. 118–132) | Generation of test data so that the entire control ﬂow path within a component in the program code (white box test) or a path through the business process (black box test) can be run through from start to finish. It must be ensured that multiple loops are also tested.  It is often required that a loop is run through zero times, once and twice. | Business end-to-end test through the entire business process (black box test) or through a method or system component (white box test), examining the system behavior specifically during the loop run. |
| Decision table (see Liggesmeyer 2002, p. 79f.). | Generation of function test data that can be used to examine the evaluation of all relevant conditions in the program code (white box test) or in the specified business rules (black box test). | Examining system behavior during the evaluation and implementation of business rules (black box test, white box test), as well as complex control decisions; particularly relevant if decision tables were used in the speciﬁcation. |
| Exploratory testing | Manual test data generated by feel and experience from which the tester believes they can identify potential errorscan, be very efﬁcient if there is a wealth of experience in testing the relevant system class or the specific system after further development. The weakest technique in methodical terms, and not suitable for satisfying formal test criteria. | Often conducted by customers or professional users without knowledge of test data generation methods. Used when there is a great deal of experience with the system and where there are always problems with specific functions or input data. |

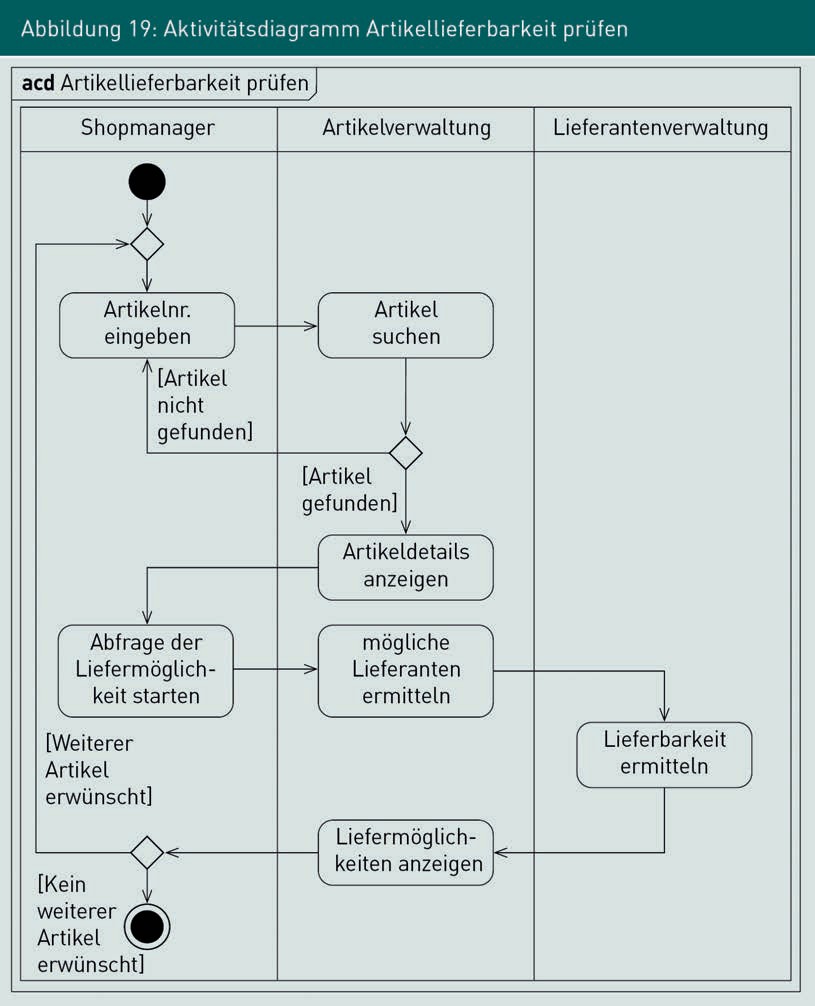
Dynamic Quality Assurance: Testing

### Use-Case-Based Test Case Generation

In use-case-based test case generation, the test cases are derived from the functional (business) use cases that are documented with use case diagrams, for example. At least one test case should be generated for each use case. The use case diagram in Figure 18 below would therefore result in at least five test cases.



Use cases often include several complex business processes that are specified with activity diagrams as shown in Figure 19 below, for example. In this instance, analyzing the activity diagram enables different test cases to be derived for the use case “Check item deliverability”. In situations where no UML diagrams have been generated, the descriptive text for the use case must be analyzed.



###### Procedure for Test Case Generation

Since it is often not possible to perform in-depth tests of all a system’s use cases, particularly relevant use cases must be identiﬁed. Criteria for the selection of use cases to be tested in detail include the following:

* + Value contribution: how much does this function contribute to the value chain?
  + Usage frequency: how many users access the function and how often?

Dynamic Quality Assurance: Testing

* Damage potential: what damage can result from unidentiﬁed errors in the function?
* Typical errors: which functions frequently contain errors in their implementation? Where are the typical weak points in comparable legacy or competitor systems?
* Required test coverage: what level of test coverage is actually required for the functional level?

These criteria can also be used to analyze the functions of complex use cases: functions that are important from a business point of view are tested intensively, while only a minimum of test coverage is required for less important functions.

Example:

It is decided for a project that use cases are to be divided into two categories according to their relevance: important use cases, whose functions are tested as intensively as possible, and less important use cases, for which all of their speciﬁed business functions are only required to be executed at least once during the test.

###### Test Coverage

The following coverage criteria, already mentioned above, can serve as criteria for the required test coverage of functions of a use case:

* Instruction coverage: each function is accessed at least once.
* Branch coverage: each outgoing control flow is run through at least once.
* Condition coverage: each condition is evaluated at least once at TRUE and once at FALSE.
* Path test: each possible path through the use case is run through completely (“in one piece”), taking possible loops into account.

The determination of the required test cases according to the required test coverage is made much easier if an activity diagram for a use case, as shown in Figure 18 above, or a comparable representation of the processes of a use case, is available. The functions that must be executed within a test case, and the order in which this occurs, can be read directly from the flow chart.

### Equivalence Partitioning and Boundary Value Analysis

###### Equivalence partitioning

Equivalence partitioning is a technique for selecting specific test data when generating test cases at the function level. All possible input values that can be assigned to a function are divided into *equivalence classes*. The term equivalence class comes from algebra, where it approximately means *a set of similar elements*.

With respect to the generation of test cases, this means that all input values that result in a functionally identical system behavior are grouped into an equivalence class. When conducting the tests, not all possible input values must be tested: instead, only one arbitrarily selected representative from each respective equivalence class need be used. With this approach, the number of test data required per test case can be reduced quite significantly. Equivalence partitioning can generally be used for all tests in which data is transferred to the system or system components (for example, in GUI tests or technical interface tests) or in which functions are accessed (in white box testing).

###### Examples for the Use of Equivalence Classes

There are one or more conditions that must be met:

The accessibility or the execution of functions depends on the conditions that must be fulfilled for business objects.

Example:

Only permanent employees are allowed to book conference rooms.

Possible equivalence classes:

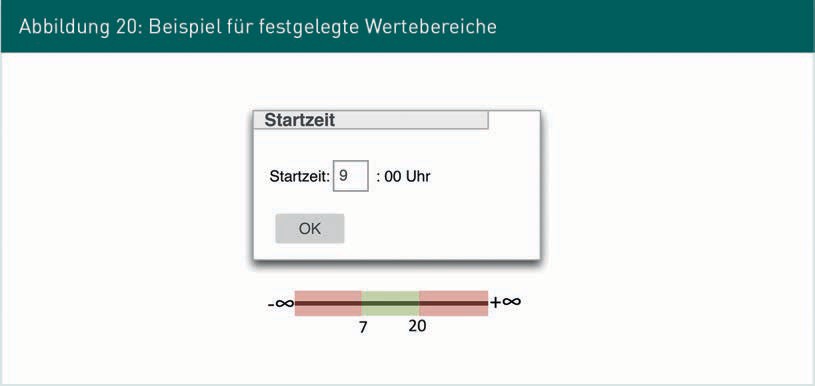
* + Valid equivalence class A1: all permanent employees
  + Invalid equivalence class A2: all freelance and all external employees

The test to determine whether access is only possible for permanent employees is not conducted with all possible data records for permanent employees, but rather with exactly one data record. To test whether it is impossible for non-permanent employees to make a booking, exactly one record is selected from the set of possible non-permanent employees. In this way, the formation of equivalence classes significantly limits the amount of test data required.

Fixed Value Range

A function expects input values in determined value ranges. Figure 20 below shows an example of a (simplified) input dialog in which a numeric value between 7 and 20 is expected for a start time on the hour.

Dynamic Quality Assurance: Testing

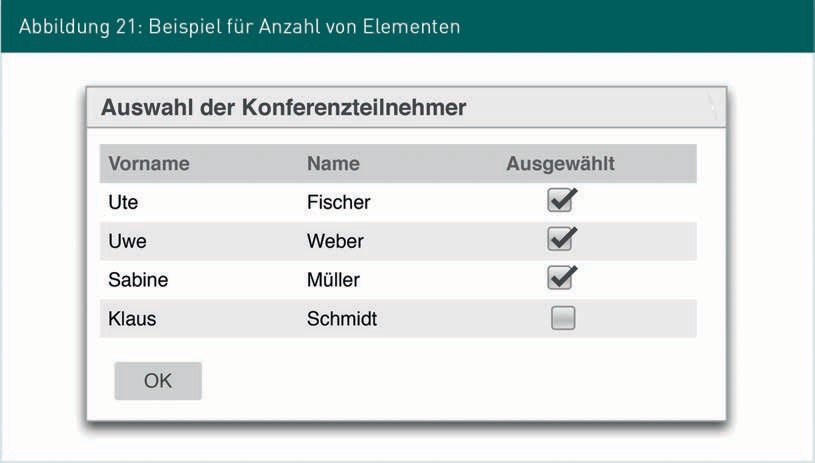


There are three equivalence classes for the design of the test cases:

* Valid equivalence class B1: all numbers between 7 and 20: {x|7 ≤ x ≤ 20 and x is an integer}
* Invalid equivalence class B2: all numbers less than 7: {x|x < 7}
* Invalid equivalence class B3: all numbers greater than 20: {x|20 < x}

Number of Elements in a Set

In a function, only a very specific number of elements is allowed within a determined input set. Figure 21 below shows an example of a dialog screen in which the conference participants are specified for booking a conference room. A minimum of three and a maximum of 25 participants must be selected.



Equivalence classes for valid values:

* + Equivalence class C1: 3 ≤ |participants| ≤ 25; all entries with any number between 3 and 25 selected.

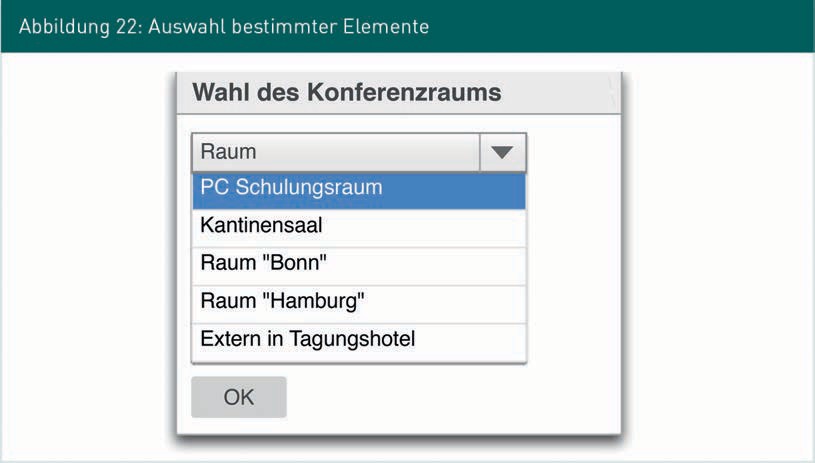
Equivalence classes for invalid values:

* + Equivalence class C2: |participants| < 3; all entries with less than 3 participants selected.
  + Equivalence class C3: 25 < |participants|; all entries with more than 25 part-

participants selected.

Selection Elements of a Set, Each of Which is Treated Differently

Values can be selected in a function, each of which triggers a different system behavior or different specialist functions. Figure 22 below shows a GUI dialog in which a specific conference room can be selected for a conference. The subsequent steps for preparing and reserving standard conference rooms behave the same with respect to the system, but all the particular rooms require different procedures in each case. In particular, the Canteen Hall may only be booked for specific occasions, but these are not yet supported by the system.



The following equivalence classes result for this example:

* + Valid equivalence class D1: {“London” Room, “Paris” Room}
  + Valid equivalence class D2: {PC Training Room}
  + Valid equivalence class D3: {Off Site in Conference Hotel}
  + Invalid equivalence class D4: {Canteen Hall}

Dynamic Quality Assurance: Testing

Note

The selection of invalid elements is used here for illustrative purposes and can usually be disabled when implementing the GUI.

###### Difference from Equivalence Classes in Mathematics

In a departure from the concept of equivalence classes in mathematics, test data as elements of equivalence classes do not need to be disjoint. Example (also see Figure 23 below): equivalence class A = {0,1,2,3,4,5,6,7,8,9 | set of all digits}, equivalence class B

= {0,2,4,6,8 | set of all even digits}, equivalence class C = {1,3,5,7,9 | set

of all odd digits}. Equivalence class A and equivalence class B have common elements, so they are non-disjoint.



###### Procedure for Test Case Generation

The procedure for generating test cases with the use of equivalence partitioning is carried out in the steps below:

Step 1

First, the equivalence classes for each input parameter of the function or each GUI element in the dialog screen are determined and clearly identified. If necessary, the result of Step 1 can be documented in a table for clarity. Table 9 below shows the result of Step 1 from the example of booking conference rooms.

|  |  |  |
| --- | --- | --- |
| Table 9: Example – Set of all Equivalence Classes for Conference Room Booking | | |
| Input Parameters | Valid Equivalence Classes | Invalid Equivalence Classes |
| Permanent position | A1 | A2 |
| Start time | B1 | B2, B3 |
| Participant selection | C1 | C2, C3 |
| Conference room selection | D1, D2, D3 | D4 |

Step 2

Test cases that cover all valid equivalence classes are generated. The number of test cases required should be kept as low as possible. As a consequence, the combination of test data should be chosen in such a way that as many equivalence classes as possible are tested through as few test cases as possible. Table 10 below shows a possible combination of three test cases that cover all valid equivalence classes for the example.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 10: Example – Test Cases for Valid Equivalence Classes | | | |
| Input Value | Test Case 1 | Test Case 2 | Test Case 3 |
| Permanent position | A1 | A1 | A1 |
| Start time | B1 | B1 | B1 |
| Participant selection | C1 | C1 | C1 |
| Conference room selection | D1 | D2 | D3 |
| Tested valid equivalence classes | A1, B1, C1, D1 | A1, B1, C1, D2 | A1, B1, C1, D3 |

Dynamic Quality Assurance: Testing

In principle, each equivalence class should only be tested once, because repeating tests with the same data does not provide additional insight. In Test Case 2 and Test Case 3, however, the already-tested equivalence classes A1, B1 and C1 are used repeatedly, since they are needed for the first execution of D2 and D3.

Step 3

Test cases are generated to cover all invalid equivalence classes. In contrast to the test cases for valid equivalence classes, the test cases for invalid equivalence classes only test one invalid equivalence class per test case. All other test data is taken from valid equivalence classes. Thus, the relevant invalid equivalence class can be more easily identiﬁed in the case of a failed test. If several invalid equivalence classes were tested in one test case, it cannot be said with certainty which of the invalid equivalence classes is the cause if a test fails. Table 11 below shows the test cases for invalid equivalence classes for the conference room example.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Table 11: Example - Test Cases for Invalid Equivalence Classes | | | | | | |
| Test Cases | # 4 | # 5 | # 6 | # 7 | # 8 | # 9 |
| Permanent position | A2 | A1 | A1 | A1 | A1 | A1 |
| Start time | B1 | B2 | B3 | B1 | B1 | B1 |
| Participant selection | C1 | C1 | C1 | C2 | C3 | C1 |
| Conference room selection | D1 | D1 | D1 | D1 | D1 | D4 |
| Tested invalid equivalence classes | A2 | B2 | B3 | C2 | C3 | D4 |

Step 4

Finally, specific input data is generated for each test case based on the selected equivalencies. A total of nine test cases were identiﬁed for testing the function for booking the conference room. In Table 12 below, a specific test data set has been generated as an example for each test case, with which the function was fully tested with regard to the identiﬁed equivalence classes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 12: Example – Specific Test Data (1) | | | | | |
| Test Cases | # 1 | # 2 | # 3 | # 4 | # 5 |
| Permanent position | Permanent employee | Permanent employee | Permanent employee | External employee | External employee |
| Start time | 8 | 8 | 8 | 8 | 3 |
| Participant selection | 10 | 10 | 10 | 10 | 10 |
| Conference room selection | “London” Room | PC Training Room | Off site in Conference Hotel | “London” Room | “London” Room |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table 12: Example – Specific Test Data (2) | | | | |
| Test Cases | # 6 | # 7 | # 8 | # 9 |
| Permanent position | Permanent employee | Permanent employee | Permanent employee | Permanent employee |
| Start time | 24 | 8 | 8 | 8 |
| Participant selection | 10 | 1 | 30 | 10 |
| Conference room selection | “London” Room | “London”  Room | “London” Room | Canteen Hall |

Evaluation of Equivalence Partitioning

Advantages

Equivalence partitioning is a simple technique for determining test data and test cases. It can be used for any technical or business function that anticipates input data. Its tabular representation enables test cases to be both generated with ease and documented intuitively.

Dynamic Quality Assurance: Testing

Disadvantages

However, business dependencies between individual equivalence classes are not taken into account; each equivalence class is essentially independent. Furthermore, equivalence partitioning is not suitable for components or systems whose behavior depends on deﬁned internal states, for example, the states of business objects or system components.

###### Boundary Value Analysis

Boundary value analysis, like equivalence partitioning, is a technique for selecting specific input data. The underlying assumption of boundary value analysis is the observation that software errors often lie in the boundary region that represents the transition from valid to invalid input values, i.e., at the boundaries of equivalence classes. Therefore, boundary value analysis is often used in combination with equivalence partitioning. When generating test cases with boundary value analysis, valid and invalid values that lie at the boundaries of equivalence classes are specifically considered. In concrete terms, the valid value closest to the boundary and the invalid value closest to the boundary are determined and used as input values for the test cases.

Example:

In Figure 20 above, the following equivalence classes were identiﬁed:

* Valid equivalence class B1, B: all numbers between 7 and 20: {x|7 ≤ x ≤ 20 and x is integer}.
* Invalid equivalence class B2: all numbers less than 7: {x|x < 7}.
* Invalid equivalence class B3: all numbers greater than 20: {x|20 < x}.

There are two boundaries from valid to invalid equivalence classes here: the boundary from B1 to B2 and the boundary from B1 to B3. Table 13 below lists all relevant boundaries that were identiﬁed in the boundary analysis.

|  |  |  |
| --- | --- | --- |
| Table 13: Example—Boundary Values | | |
|  | Valid Boundary Values | Invalid Boundary Values |
| B1 to B2 boundaries | 7 | 6 |

|  |  |  |
| --- | --- | --- |
|  | Valid Boundary Values | Invalid Boundary Values |
| B1 to B3 boundaries | 20 | 21 |

When combining equivalence partitioning and boundary value analysis, in addition to the boundary values, one representative of each equivalence class is often included in the set of test data as test data that lies in the middle value range, away from the boundary. In contrast to equivalence partitioning, boundary value analysis expands the set of test data to be considered and thus also the set of test cases to be conducted. However, if values of an equivalence class need to be used in several tests to achieve the required test coverage, different boundary values can be used in order to keep the total number of necessary test cases as low as possible.

Example:

In the test cases shown above in Table 11 and Table 12, a value of equivalence class B1 is required a total of seven times. In this case, both the boundaries 7 and 20, as well as values from the middle value range, such as 12 or 14, can be used as representatives of B1. In this way, the valid boundaries can be tested without additional effort.

### State-Based Test Case Generation

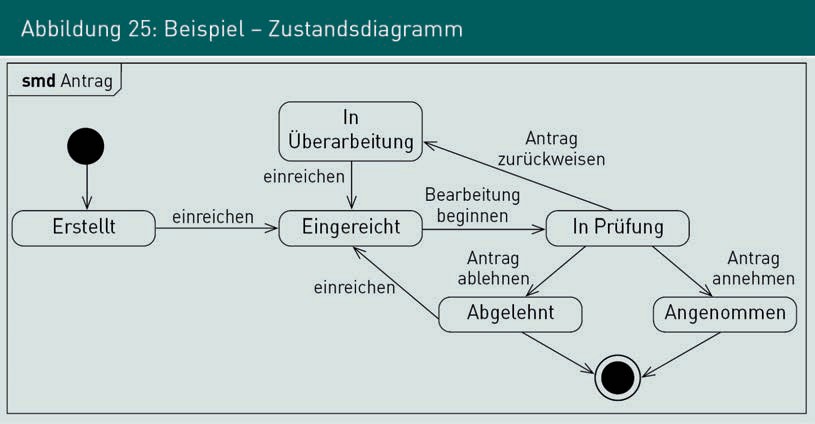
State-based test case generation is used for systems or system components whose behavior is significantly controlled by an internal functional or technical state. Examples of functional states are the phases in the life cycles of business objects (such as: requested, accepted, rejected) or processes (such as: opened, expert opinion commissioned, damage assessed, damage settled). Examples of technical states include the internal states of system components or protocols at technical interfaces. Therefore, state-based test case generation can be used for white box testing, as well as for black box testing. Test cases can be derived relatively easily on this basis, particularly if state diagrams are used as part of the speciﬁcation. In principle, state-based test case generation is used to create test cases that examine the achievement of deﬁned states via the state transitions provided for this purpose. The non-support of unforeseen state transitions by the system is also explicitly examined.

###### Procedure for Test Case Generation

Test cases are generated based on the speciﬁed states and state transitions. They can be speciﬁed using state diagrams, state tables, text, or a hybrid approach. Figure 25 below shows an example of a state diagram

Dynamic Quality Assurance: Testing

which is part of the speciﬁcation and on the basis of which test cases are to be generated. If no state diagrams or state tables were used in the speciﬁcation, they may need to be derived from the speciﬁcation when generating test cases.



Step 1: create a state transition table

The focus of a state diagram is the graphical representation of the desired system behavior. Undesired behavior is not explicitly speciﬁed in the state diagram, but must be explicitly excluded when conducting software tests. Therefore, for test case generation, undesired behavior must also be explicitly described. For example, tests ensure that no single function in the system causes a state transition from

“Rejected” to “Accepted”. For easy identiﬁcation of allowed and disallowed transitions, the state diagram is transformed into a state transition table. Figure 26 below illustrates the elements of a typical state transition table: one column and one row are created for each deﬁned state in the diagram.

Step 2: fill in the table

Valid combinations of states and transitions in the diagram lead to a subsequent state. In each case, the name of a transition is noted exactly in the cell whose initial state is provided by the column and whose target state is provided by the row.

All cells that remain empty after the transitions deﬁned in the diagram have been transferred are state transitions that the system is not allowed to execute. For example, they are noted with “–” in the table. In Figure 26, no transition to the “Created” state is possible from the “Submitted” state. In contrast, the “Submit” transition in the “Created” state leads to the subsequent “Submitted” state.

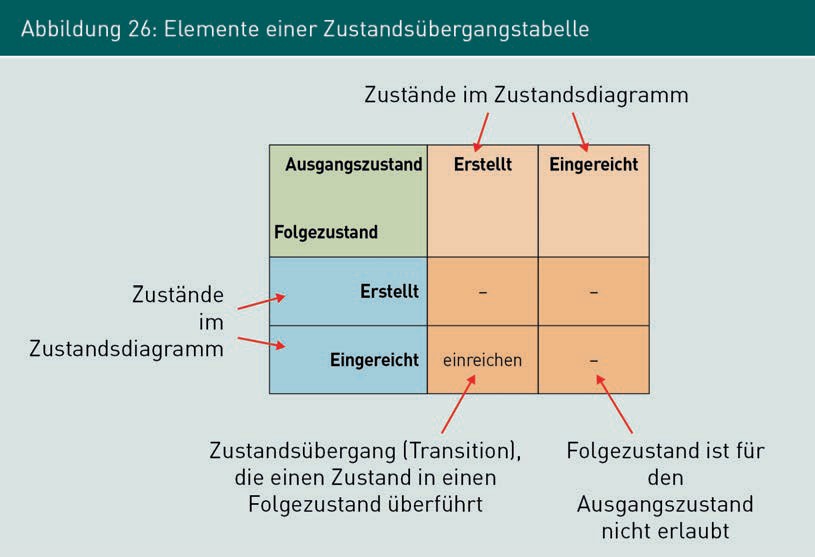
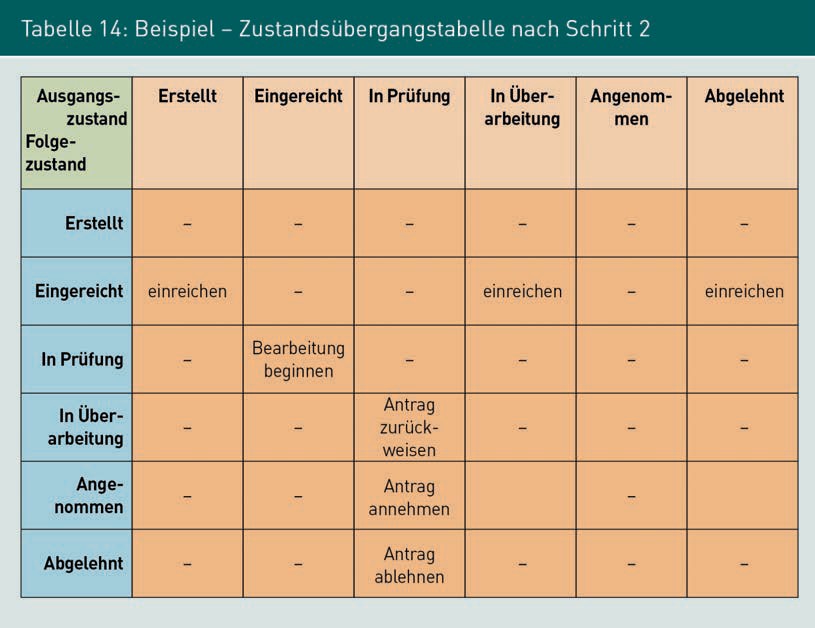


Table 14 below provides below a complete state transition table for the state diagram shown in

Figure 26 above.



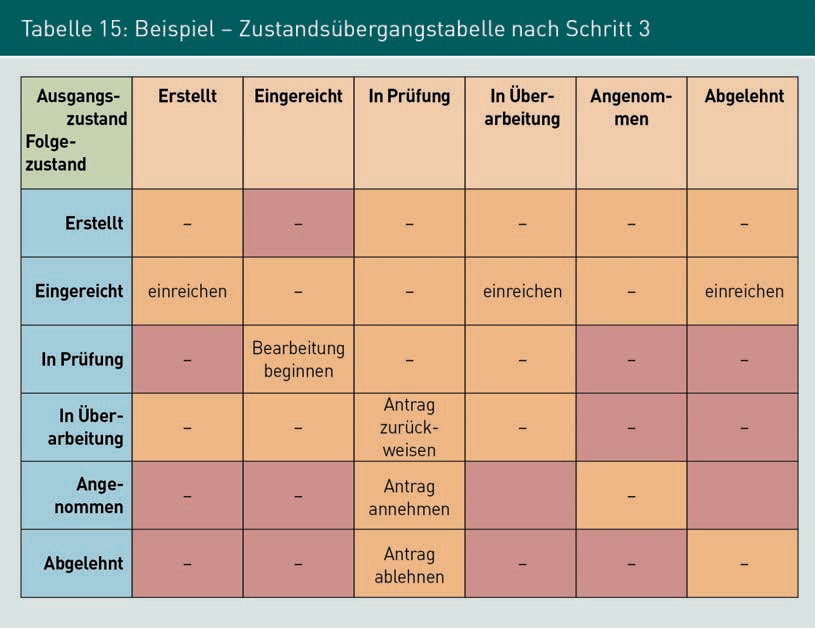
Dynamic Quality Assurance: Testing

Step 3: enter error states in the table

Depending on the size of the state transition table and the application of the system under testing, the disallowed transitions (marked with “-” in the table) can be divided into two categories:

* Category 1: error situations whose occurrence leads to functional errors and are therefore to be avoided at all costs. Example: a transition from the “Created” state to the “Accepted” state, since a request must be examined in every case before a decision is made.
* Category 2: transitions that are disallowed but are trivial and, if they are carried out, do not lead to functional misbehavior. Example: a transition from the “Submitted” state to the “In Revision” state, since no functionally important decision is overlooked here.

The categorization is relevant for the derivation of test cases so that test case generation can concentrate on real error situations. Disallowed transitions that do not lead to a functional error are only tested with low priority. The assignment of the disallowed transitions to the categories in question requires expert knowledge and must be coordinated with the business-oriented stakeholders, if necessary. Table 15 below shows the state transition table with error situations marked in red. The non-implementation of these situations must always be taken into account when generating test cases.



For example, it must be ensured that the “Accepted” and “Rejected” final states can only be reached from the “In Revision” state. However, if one of the two states has already been reached, a transition with the same subsequent state as the initial state does not lead to any functional errors.

Step 4: derive test cases from the table

Test cases can now be derived from the state transition table with marked error states available after Step 3. The outcome of test case generation is a set of sequences of transitions that can be used to test the functions to be implemented by the system, as well as the transitions that are not to be supported by the system. Depending on the size of the table and the requirements for the test cases, the following different levels of test coverage can be distinguished:

* + All states: with respect to the state diagram, this test coverage is the minimum objective. For this purpose, sequences of transitions are determined with which each deﬁned state is reached at least once.
  + All allowed state transitions: at this level of test coverage, all transitions deﬁned in the state transition table have been executed at least once.
  + All error situations: in addition to all allowed state transitions, all error situations are also tested for non-implementation.
  + All disallowed state transitions: in addition to the error situations, all disallowed state transitions that are not error situations are also examined for non-implementation.

Figure 27 below illustrates the dependencies of the coverage levels for state-based tests: For example, the testing of all states is included in the testing of all allowed state transitions. The testing of all disallowed state transitions, alongside the testing of all of all error situations, also includes the testing of allowed state transitions:

All states ⊂ All allowed state transitions ⊂ All error situations ⊆ All disallowed state transitions.

Dynamic Quality Assurance: Testing



###### Evaluation of State-Based Tests

Advantages

State-based test case generation is very easy to apply on the basis of speciﬁc state diagrams, which is why it is widely used in practice. The state transition table enables the test cases to be easily derived, the test coverage to be visualized and, for example, the current progress of the test execution, as well as the test result, to be displayed by coloring table cells. If the dialog flow of GUIs is speciﬁed with state diagrams, state-based test case generation can also be used for generating GUI tests.

Disadvantages

If a large number of states to be tested are identiﬁed, the state transition table quickly becomes very large and confusing. In business-oriented state diagrams, only the functions that represent a real state transition are often noted in the state diagram. In general, there are also business-oriented functions that do not change the business-oriented state. Under certain circumstances, these functions are forgotten during test case generation. Examples of this are functions for printing or editing data stored in the system.

### Generation of Random Test Data

When generating random test data, test data that is used as input values for functions under testing is generated. For example, this technique is used for testing user interfaces with many input elements or for testing data exchange via technical interfaces. The basic principle of test data generation is based on the fact that valid values are randomly generated within a given range of input parameter values. For example, the boundaries

of value ranges can be determined with equivalence partitioning and the specific input values can be defined using random test data. Depending on the requirements of the test cases, complex random test data can also be generated on the basis of existing test data sets. In this case, a given, non-randomly generated set of test data is randomly combined according to a defined principle. This ensures that the basic structure of the test data is as real as possible, but without using exactly the same data in every test case.

Example:

The complex input dialog of a web application is to be tested. This dialog requires various inputs, including addresses. For this purpose, a database is built with 50 valid postal codes, city names, street names, and house numbers. By randomly combining this data, 504 different test addresses can be generated, which corresponds to a total of 6,250,000 variants.

###### Evaluation of Random Test Data

Advantages

In combination with equivalence partitioning, random test data generation is a technique where many different test data sets can be generated with comparably little effort. Once a suitable test data generator exists, any number of test data sets can usually be generated. The automatic generation ensures that each test data set has the required attributes and that human errors arising from manual test data generation can be excluded.

Disadvantages

The random generation of test data creates a great deal of different, but not truly realistic, data in every case. For example, if the address data from the above example is to be automatically checked with address validation, randomly generated but unrealistic test data will not help. Therefore, before the test data generator is created, the requirements for the structure and content of the test data must be speciﬁed in detail. Whether the effort and expense required for this is in reasonable proportion to the benefit must be decided individually for each project.

The generation of random test data is therefore often a useful addition to other techniques, but is usually not sufficient as the only technique.

Dynamic Quality Assurance: Testing

Summary

Dynamic test procedures (also: software tests) are analytical QA measures in which a system or parts of a system are executed. The various techniques are distinguished by the way in which the test cases are generated, which enables a distinction to be made between black box tests and white box tests. However, since not all possible combinations of input data can be tested practically, a set of test cases and test data that are as suitable as possible and achieve the highest possible test coverage with the least possible effort and expense must be identiﬁed.

With use-case-based test case generation, the test cases are derived from the functional use cases that, for example, are documented with use case diagrams.

Equivalence partitioning is a technique for selecting specific test data on the level of functions. All input values that result in a functionally identical system behavior are grouped in an equivalence class.

In state-based test case generation, test cases are created based on a state diagram or a state transition table. In this process, the non-support of unforeseen state transitions by the system is also explicitly examined.



# Unit 6

## Systematic Testing of Software

#### STUDY GOALS

After completing this unit, students will know ...

... which activities for methodical testing can be described in general for each test level.

... what component tests are and how they can be supported by automated unit tests and test-driven development.

... what integration tests are and which integration strategies are available.

... what system tests and acceptance tests are and which different types of focus these tests each have.

DL-D-IQSS01–L06

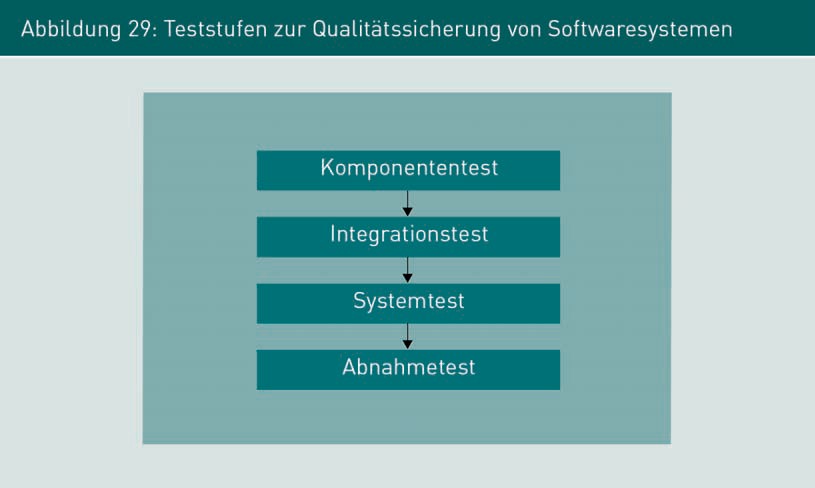


### Introduction

Depending on the current progress of a project, software is tested at different test levels: component tests, integration tests, system tests, and acceptance tests. Firstly, this unit introduces the activities that must be carried out for methodical testing on all test levels. The individual test levels are then described in detail and specific techniques and approaches are presented.

### Methodical Testing Activities

*Test levels* are distinguished for testing a software system that has been created. As already mentioned, it is too risky to wait until implementation is complete before conducting software quality assurance. Therefore, in industrial software development, test activities are started as soon as the first software files have been created. Figure 29 below shows a schematic overview of the various test levels in software engineering.

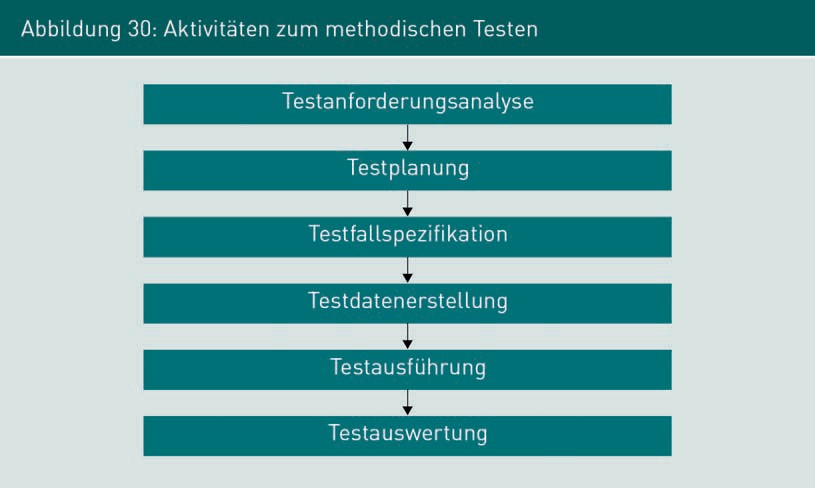


Component tests are often conducted by the developers themselves in order to test the functions and system components they have created in isolation from the overall system. Integration tests specifically test the interaction between several components that are connected to each other via technical interfaces. The entire system is tested as part of the system tests. Here, in addition to testing functional processes throughout the entire system, system behavior is also tested under extreme conditions with regard to compliance with the stipulated quality requirements.

Systematic Testing of Software

The acceptance tests described are conducted by the customer, who accepts the software system on the basis of the results.

The activities shown in Figure 30 below are identiﬁed for the methodical execution of software tests and must be conducted for each test, regardless of the test level. However, these activities vary in scope depending on the complexity of the test to be conducted. For example, test planning for component tests is usually carried out only very briefly, while planning for a comprehensive system test can take several days or weeks.



###### Test Requirement Analysis

As described above, industrial information systems cannot be fully tested. However, the effort and expense expended to prevent errors after delivery of the system should be as cost-optimized as possible. Therefore, the risks remaining due to undetected errors must be determined individually for each system. The risks identiﬁed must be addressed through the design of specific QA measures. Test requirements analysis is roughly comparable to the requirements analysis of the software system: here, the quality objectives or quality level to be achieved are determined and the system functions and system attributes to be tested are described. The outcome of the test requirements analysis is still a very rough description of the test cases or the test object without going into detail regarding the process or the test data.

Example:

“Test if the conference room can only be reserved by authorized people” or “Test the request submission when it is simultaneously used by 2,000 users”.

###### Test Planning

The tests are planned after the requirements analysis. The more complex the preparation, execution, and evaluation of a test, the more carefully the test planning must be conducted. Typically, this activity is most pronounced for system tests. If the complete system is to be tested in an environment as close to production as possible, it is necessary to set up a complex test environment. In practice, it is not uncommon for test environments to take several days or weeks to deploy. Depending on the technical or personnel resources required for conducting the test, these resources must be planned and their availability ensured.

Example:

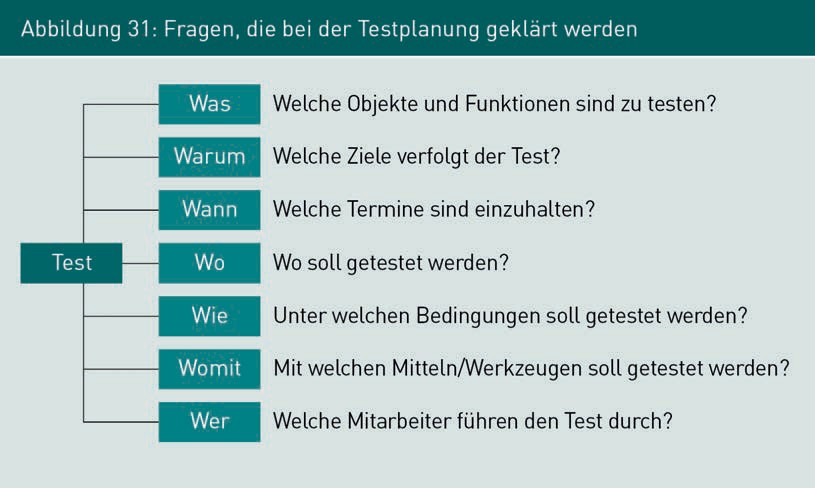
Depending on the contract with the supplier, the payment for hardware use is often based on the resources actually provided for the demand in question (CPU, memory, data transfer). If test cases are executed at the same time as complex business functions, this can lead to higher resource utilization and thus to a high additional payment for the provision of the infrastructure. If this is not taken into account during test planning, the costs for a test case can amount to several tens of thousands of euros.

Example:

A large number of simultaneous users is required for a load test that is to be as close to reality as possible. If an automatic test cannot be carried out, as many employees as possible must simultaneously carry out predefined test cases at a specific time. To do this, all those involved must first meet the technical requirements for the test (for example, having the necessary hardware and software or being able to log in to the system) and then they must also be trained to carry out the test.

The following questions must therefore be answered in the course of test planning:

Systematic Testing of Software



###### Test Case Speciﬁcation

In the course of test case speciﬁcation, the specific test cases are generated and described before subsequently being executed. During test case speciﬁcation, a detailed description of the following aspects is created based on the following test case requirements:

* purpose of the test case,
* test objects,
* state of the system before and after test execution,
* type and structure of the test data,
* required hardware and software environment,
* required personnel, and
* dependencies on other test cases.

For small and simple test cases, the test data is often generated at the same time as the test case speciﬁcation. For complex tests, such as system tests, the type and structure of the test data are described during test case speciﬁcation, but they are generated as a separate activity. Before starting test case speciﬁcation, how the test cases are to be obtained and what degree of coverage is to be achieved with the test cases should be determined.

Table 16 below provides an example template for the component testing specification that can be used for test case specification and documentation.

|  |  |
| --- | --- |
| Table 16: Example Template for Test Case Speciﬁcation | |
| Template for Specific Test Cases | |
| Test Case ID | Unique ID |
| Test object | Ofﬁcial name of the system component or interface under test, or of the function of the system/component. |
| Tested detail | Name of the class(es) or component(s) that will be examined in this test. |
| Function(s) that are tested | Detailed name of the method/functions/interface and the relevant parameter list. |
| Brief description of the test | Brief description of what is to be examined or ensured with the test. |
| Test data/input | Which input data is needed to run the test case? With which parameters should the functions to be tested be accessed? If they are objects, which attributes should they include? If they are files, what are their contents? |
| Expected result/ expected system reaction | Which outputs or system reactions are expected from the test data? |
| Preparation/prerequisite | Specific steps to prepare the system/component for the test:   * Generate or import test data into the database. * Access defined methods so that the system is in the state in which the test can be executed (e.g., users logging onto the system). * Ensure that defined data is available in the data bank. |

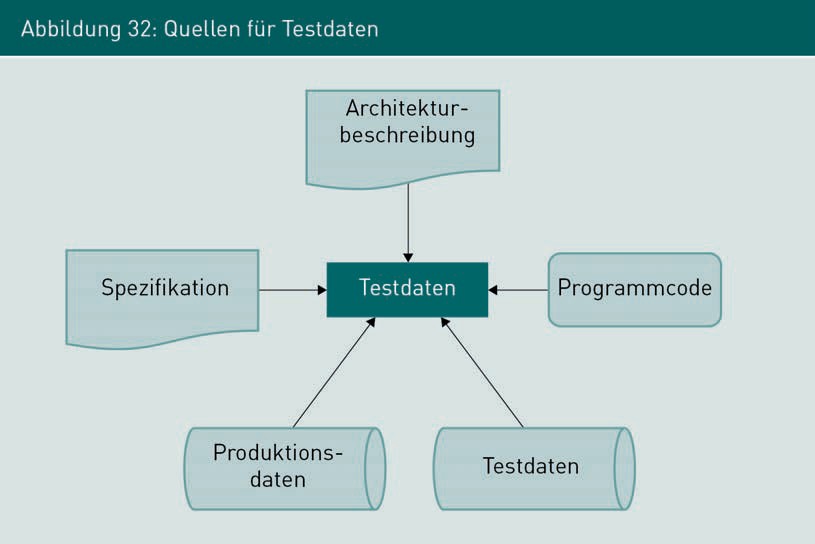
Systematic Testing of Software

|  |  |
| --- | --- |
| Template for Specific Test Cases | |
| Test Case ID | Unique ID |
| Execution | Which system function(s)/method(s) must be executed during the test?  For more complex test cases with multiple accesses: in which order must the methods/system functions be executed? |
| Review/post-condition | Provide the specific conditions of a passed test here.  Which system behavior/outputs are expected from the system?  Which criteria must the data stored in the system meet?  Which criteria must be met for the test to be successful? How are the criteria measured? |

###### Test Data Generation

The activities of test data generation are closely linked to the activities of test case speciﬁcation and they often cannot be clearly distinguished from each other. Test data generation results in the test data sets that are needed to execute the test cases specified. How much test data is needed and what requirements there are for the test data are defined in the test case speciﬁcation. For simple components or integration tests, simply structured test data is often sufficient. For business end-to-end tests during a system test or an acceptance test, complex test data sets are often required, which must specifically fulfill special functional conditions. Therefore, detailed expert knowledge is usually required for test data generation.

Various techniques for generating test cases have already been explained. When generating test data, it is also important to ensure that the test data used is as real as possible. Only real data can be used, for example, to examine whether the size of GUI elements is sufficiently dimensioned for the input and output of data. All sources available to the tester can be used to generate test data. Figure 32 below shows typical sources: In addition to the specification and architecture descriptions, the program code of the system, as well as existing production data or the existing test data inventory, are also suitable.



The targeted generation and maintenance of a test data inventory can be worthwhile if it is foreseeable that the same test data will be used in several tests. This inventory of data can be continuously expanded to include special cases. It should also contain data sets whose structure has always caused problems during automatic processing. A well-maintained test data inventory can ensure that typical sources of error are detected quickly and reliably.

###### Test Execution

Ticket systems, referred to as bug trackers, are systems to support the documentation, evaluation, and correction of software errors.

Conducting software testing includes the execution of the speciﬁed test cases using the test data sets and the logging of the results. The logged results are used to decide whether an acceptable level of quality has been achieved for the current project situation. If errors were identiﬁed during test execution, they must be documented in a deficiency list. **Ticket systems** or bug trackers are used for this purpose, in practice.

The tests are conducted either automatically or manually. In the case of manual execution, human users create or ensure the preconditions for starting the test case, execute the test, examine the post-conditions, and then log and evaluate the system behavior.

When software tests are automatically executed, the test case speciﬁcation is described in conjunction with the test data in such a way that the tests can be automatically executed by test systems. Automatic tests are fundamentally possible at all test levels.

Systematic Testing of Software

This applies to technical tests in which program functions are accessed either directly or via a technical interface, but it also affects GUI tests in which values are entered into a user interface and deﬁned interaction steps are executed.

###### Test Evaluation

After the tests have been carried out, the results are evaluated and the quality level of the system is assessed. While the evaluation of automated component tests, for example, is automatically generated when the test execution is completed, the individual results must first be compiled and aggregated for the evaluation of manual system testing with several dozen testers. The results of the evaluations form the basis for decisions regarding further activities in the software process: for example, if the targeted quality level was achieved, the quality gate can be completed or the system can be delivered. If the targeted quality level has not been achieved, the necessary rework must be carried out.

### Component Tests (also: module tests, unit tests)

To enable a software system to be implemented by several software developers, it is broken down into its logical components (also: components, modules, classes, building blocks). Each component is implemented separately and brought into the system after completion (also: integrated). Compliance with speciﬁed requirements can be reviewed during or after completion of a software component. The isolated testing of individual software components is called component testing.

Examples:

* Premium calculation: calculation of insurance premiums depending on the selected benefits.
* Shopping cart: calculation of the total sum of all items.
* Address component: generating the correct address from the customer data for correspondence.

###### Test Automation with Unit Tests

For most programming languages used in industrial software development, there are frameworks that support the automated execution and evaluation of unit tests. For example, there is the JUnit framework for the Java programming language. Unit tests are generated and executed by developers to test the quality of the program code they have created themselves. This type of test is used to examine the correct execution of individual functions of technical classes and components.

This testing is generally carried out before a developer’s work is integrated into the existing program code.

The use of automatic unit testing aims to achieve the greatest possible test coverage of the instructions and control structures in the program code created. Unit tests are white box tests which, depending on the specific test case requirements, are intended to test all instructions, conditions or paths of a system component. Automatic unit tests are often required to have reproducibility, independence, and regressability.

Reproducibility

Once a unit test has been generated, it should be reproducible without restriction. This is achieved in several ways, including by using test data that is permanently deﬁned in the test case and does not depend on the successful execution of other unit tests.

Independence

Furthermore, unit tests should not have any functional dependencies on each other: the order in which unit tests are executed should not play a role. Each test must individually ensure that the required preconditions are established. For example, the required data inventory is imported into the database before tests are executed, with this inventory then used to execute the test. If another test is subsequently executed, all changes to the data model and the internal system state must be reverted so the original system state is restored for the execution of a new test.

Regressability

Particularly with software processes in which systems are developed in an evolutionary manner, i.e., new functions are added to the system step by step, the required level of quality must be ensured before a new system version is released at the end of an iteration. For unit testing, this means that all unit tests that have already been successfully run must also be successfully executed after a revision. The re-execution of tests that have already been passed after changes have been made to the program code is called regression testing. At the end of an iteration, all previously generated and all new unit tests must be successfully run. This is the reason why test automation is an important prerequisite for evolutionary software development.

###### Approach: Test-Driven Development

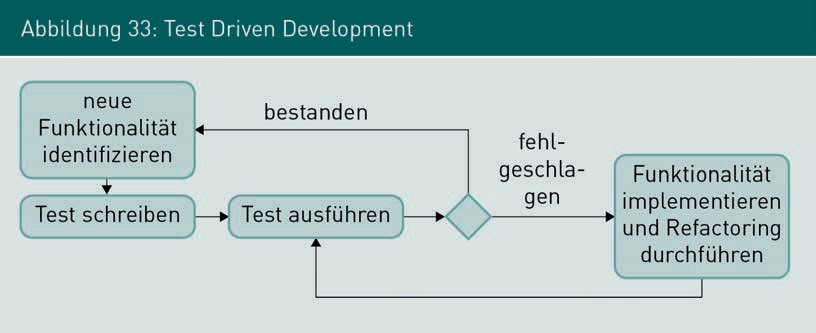
**Test-driven development** is an approach that focuses on a close interlocking of the implementation of program code and the generation of unit tests for testing the program code. Test-driven development is intended to ensure high quality and high coverage of the program code that has been created through unit tests in which the *generate test*

Systematic Testing of Software

*case* and *create program code* activities are carried out in the exact reverse order of “traditional” test case generation.

Figure 33 below shows the basic test-driven development procedure.

The crucial difference from the traditional development of test cases is the generation of unit tests on functions before their method body is programmed. After the need for a new function in a class or component has been identiﬁed, unit tests are first generated to test the new function to be created. If these unit tests are executed, they must initially fail because the relevant system functions have not yet been implemented. Their implementation is not started until after the test case has been generated. The system functions are then implemented step by step and tested during the implementation.



If it is determined during implementation that further functions are required, test cases are first re-generated before the actual functions are implemented. In practice, the activities of testing and implementation alternate in short cycles, with a cycle often lasting only a few minutes. If the test is generated before the program code is implemented, the developer must first deal in depth with the function to be created before the first line of program code is written. This need to think through the functionality and possible error situations in advance creates an awareness of potentially critical and error-prone areas of the functions that can then be taken into account directly during implementation.

Test-Driven

Development

This is an approach to generating unit tests in which the test cases are already implemented before the program code to be tested.

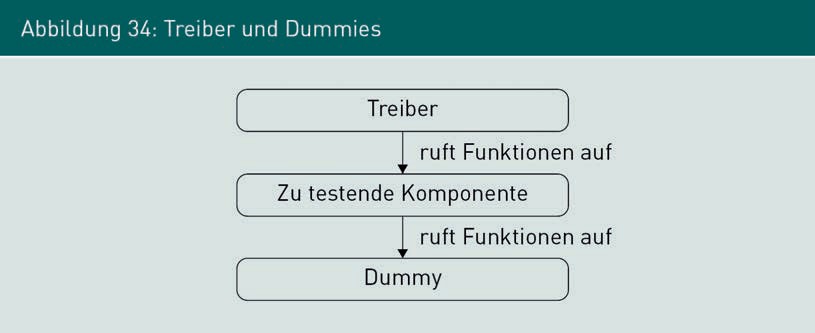
### Integration Tests

Once two or more software components have been completed, they can be combined to form a system, i.e., integrated. During or after integration, integration tests are used to test the interaction between groups of components and to examine whether the components work together as described in the speciﬁcation.

Examples:

* + Integration of an external online payment system with an online store.
  + Integration of an ordering component of a supplier in an online store.
  + Integration of an app for mobile platforms to enable orders from mobile devices in addition to the web interface.

Compared to component tests, integration tests are not intended to provide an in-depth examination of the internal behavior of individual components, but rather the interaction between the components via their technical interfaces. The relevant aspects here are the functions provided by the interface as well as the data exchanged via the interface. For integration tests, test cases must therefore be generated and executed specifically to test the functions provided by the interfaces as comprehensively as possible. In particular, this includes the behavior of interfaces in the event of incorrect accesses and incorrectly transmitted data. Apart from the focus of the test cases on the interfaces, integration tests do not fundamentally differ from component tests in the way they are generated or executed. For example, test-driven development is also possible for interfaces between system components.

Drivers These are software fragments that simulate the accessing of other components.

.

Dummies These are software fragments that simulate components and are accessed by other components.

.

Particularly in complex software systems with a large number of components, not all the components are ready at the same time. In these cases, missing components are simulated by **drivers** and **dummies** so that testing can begin. Figure 34 below illustrates the interaction between components, drivers, and dummies.

Systematic Testing of Software

The software fragments referred to as drivers simulate the accessing of other components, whereas dummies simulate components that are accessed by other components. In particular, technical interfaces to external systems are simulated in the integration test by dummies and drivers. Only the finished software system can be connected to external systems, but the interaction of the interfaces must be tested beforehand.

The integration strategy is used to determine the order in which the individual components are integrated and tested in a project. The three different integration strategies — bottom-up, top-down and by-value—are presented below.

###### Bottom-Up Strategy

In the bottom-up strategy, the **base components** are implemented first. Drivers are used intensively in this process. Building on the base components, the components that access fundamental services are then implemented. Figure 35 below shows the basic principle of the bottom-up strategy, read from left to right: Components 3 and 2 are created first, which are later accessed by Component 1 (t1). The input of the test data and/or the access of the functions of these components, which can be specifically tested, takes place via drivers. Subsequently, Component 1 is created (t2), which requires the functions of Components 3 and 2 and is accessed itself by a driver. Finally, the functions are implemented with the user interface (t3), which accesses Component 1. Only then are all the components implemented and can be fully tested without the use of drivers.

In the bottom-up strategy, system development begins at the lowest system level and is successively carried out up to the top system level. The drivers are replaced by the real components piece by piece as they are completed.

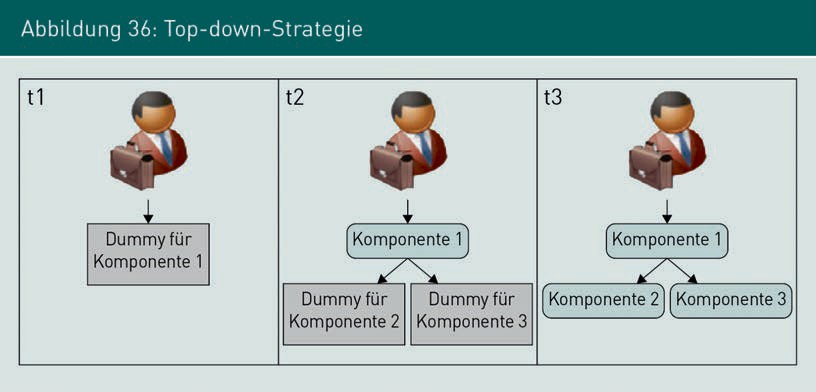
Base Components These are components that provide fundamental services within a system and are accessed by other components.



One advantage of the bottom-up strategy is the early testing of base components, which are highly heterogeneous in terms of the technologies used, due to their proximity to the hardware and infrastructure. In addition, the test data generated can be entered in a targeted and direct manner through the use of drivers. However, these advantages are offset by the disadvantage that the system can only be presented to its users at the end of the integration, which means that a significant project risk cannot be addressed at an early stage. Furthermore, no targeted false return values of accessed components can be tested since no dummies are used with the bottom-up strategy.

###### Top-Down Strategy

When using the top-down strategy, the integration of the system components starts at the top system level and ends with the integration of the base components. Figure 36 below shows an example of the basic principle of the top-down strategy: the process begins with the implementation and testing of the top system level (t1), which is the user interface, in this case. To execute the tests, a dummy for Component 1 is required to simulate its functions. Subsequently, Component 1 is used to implement the next system level (t2) and integrate it into the system. For this purpose, the dummy for Component 1 is removed and the dummies for Components 2 and 3, which are required by Component 1, are created. Finally, the base components are implemented as part of a top-down strategy and the remaining dummies are removed upon their integration into the overall system.



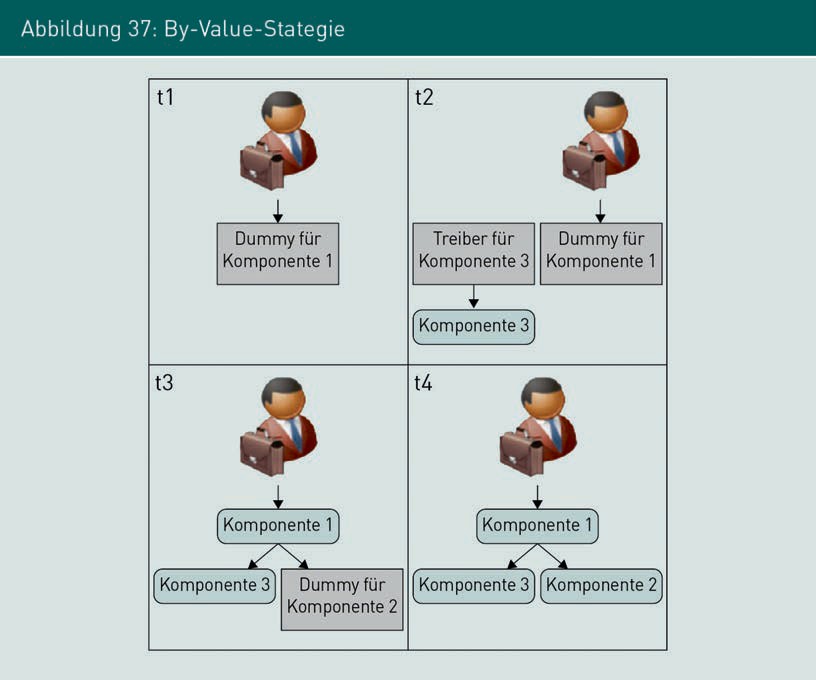
The advantage of the top-down strategy is the early availability of the system level with which the users work. This means that initial experience and improvement requests can be gathered quite quickly to increase user satisfaction. In addition, false return values of accessed base components can be easily simulated, since they can be generated directly by the dummies. One disadvantage of the top-down strategy, however, is the very late integration of base components, which often entails various technology risks.

Systematic Testing of Software

In contrast to the bottom-up strategy, the test data cannot be entered in such a targeted manner, since no drivers are used in the top-down strategy.

###### By-Value Strategy

With the integration strategy referred to in this course book as the by-value strategy, the components are integrated according to the aspects of the value orientation of individual components. In comparison to the top-down strategy or the bottom-up strategy, the by-value strategy is not strictly oriented in line with the technical level of the components in the overall system, but rather with the actual needs of the software project. The order in which the components are integrated into the total system depends on the project in question. Thus, the components with the greatest value contribution are the first to be implemented. Figure 37 below illustrates the use of the by-value strategy as an example: first, the user interface is implemented (t1) so that specific system artifacts can be shown to the stakeholders fairly quickly. A dummy of Component 1 is needed for this. Then Component 3 is implemented (t2), the realization of which comes with a high technical risk. A driver for Component 3 is needed for this. Next, Component 1 is used to implement the interconnection of the GUI and Component 3 (t3), for which a dummy for Component 2 is still required. Finally, Component 2 is implemented and the entire system is completed.



Thus, the by-value strategy is a hybrid that requires drivers as well as dummies to execute the component tests.

The advantages of the by-value strategy lie in the option to decide which components are to be integrated and tested depending on the current project situation. The targeted input of test data as well as the targeted simulation of false return values are possible with the use of drivers and dummies. The disadvantage of the by-value strategy is the use of both drivers and dummies, which means that the effort and expense required to create pure test artifacts may be higher compared to other integration strategies.

###### Discussion of the Different Integration Strategies

For the development of industrial information systems, the bottom-up and top-down strategies generally represent ideal models that are not used in practice in their pure form. In reality, the individual components are typically completed at different times and sometimes deviate significantly from the originally planned times. As a consequence of the strict specification of the integration sequence it is not possible to react to project-specific risks and developments in a targeted manner. The by-value integration strategy referred to here offers the project team the corresponding freedom according to each actual need and also to react appropriately to changed situations: if both drivers and dummies are needed to test the interaction of components,

Systematic Testing of Software

they can be created and used as required.

###### Use of Dummy Components

To support the execution of integration tests, the development team responsible for the development of a component may be required to simultaneously create and maintain a dummy for this component. This dummy provides all the interfaces and functions required by the component that can be used for test purposes. Other development teams that need the functions of this component or need to access their functions with the component for test purposes can deploy the dummy component at any time. The responsibility to maintain a dummy component ensures that all its functions correspond to the current state of development and that changes to the component interface are also maintained in the component’s dummy itself. Figure 38 below illustrates the use of dummy components. Since they are able to simulate all interface functions, they can be used both as drivers and as dummies during integration tests.



In each case, a dummy must be created for each component for the execution of integration tests. It is ensured at all times that the individual dummy behaves in exactly the same way as the real component. This means that each development team does not need to rebuild the test infrastructure for testing its components and can instead use the dummies provided by the component developers. High test quality can be ensured by having a dummy and a real component delivered from a single source.

### System Tests

Performance Tests

The behavior of the system under load is tested for compliance with the specified quality characteristics and the measurement of performance.

reviewed.

Load Tests

These test the system behavior under particularly high demands, e.g., by selectively withdrawing resources.

Once development work has been completed and all the software components have been integrated into a finished system, the system is tested as a whole with a system test. The objective of system tests is to review whether the system as a whole meets the speciﬁed requirements. In addition to function tests, this also involves **performance tests** and **load tests**. A major challenge in a system test is the replication of the customer’s productive environment in a manner that is as true to the original as possible. This includes:

* the hardware and software environment in which the system is operated,
* the software systems with which the system under testing has technical interfaces,
* a realistic utilization and realistic user behavior
* real data sets, if possible.

In particular, the provision of data sets that are true to the original and simultaneously comply with data protection regulations is a major and often unsolvable problem.

###### Function Tests

The objective of function tests within the scope of a system test is the review of the functional requirements in the overall system. In contrast to integration tests, in which the interconnection of the individual system components has already been tested, system tests focus on the user’s point of view. The system is considered entirely as a black box. Function tests are used to examine whether the system support for business processes and business functions is implemented exactly as defined by the speciﬁcation. The test cases are executed using data input in the user interface or by accessing the system interfaces through which the system is connected to its peripheral systems. The peripheral systems should not be dummies or drivers, but rather systems in the version that will exist in the future execution environment of the system. The test cases are evaluated on the basis of the externally visible system behavior, which includes the output via the user interface, the documents created, the messages returned to system interfaces, the changed data inventory of a database, or the log files generated by the system.

###### Performance Tests

Performance tests are used to examine the behavior of the system under load and to determine whether it supports the specified quality attributes. Typical scenarios of a performance test are the simultaneous usage of the system by many users or the processing of large amounts of business data. The results of the performance test can be used to make specific assertions about the behavior of the created system in its future system environment.

Systematic Testing of Software

In principle, a performance test serves to minimize risk when the system is commissioned, since it allows assertions to be made about system behavior in the subsequent execution environment. Reliable assertions can only be made once all the technical components have been integrated and the system has been installed on the hardware on which it will later run. It is important here that the hardware and software environment is reproduced exactly, since even the smallest changes can distort the results of performance tests. The test results can also be used to review whether the IT infrastructure provided by IT operations, such as storage space, RAM, computing time, or network connectivity, is sufficient. Performance tests must therefore be prepared very carefully.

In addition to the availability of the required IT infrastructure, the system must be installed in the test environment and connected to all the surrounding systems. This requires all the necessary activities for commissioning the system. Therefore, attention must also be paid to the availability of the appropriate personnel.

Typical indicators of the performance of software systems measured in a performance test include the following (Grechenig 2010, p. 328):

* + Latency: the time that elapses between a query to the system and the response; for example, the time that the system takes from clicking on the OK button to displaying the confirmation dialog when an order is placed in an online store.
  + Throughput: processing speed of functions or data, for example, the number of order transactions processed per minute.
  + Transaction rate: measure of the processing speed of deﬁned, complex changes to the data set, for example, the number of transfers posted in the core system of a direct bank.

Performance testing is also used to determine and examine the service levels that a system operator has promised in service level agreements (SLAs). Service level agreements stipulate service quality characteristics such as response times or availabilities for the software provided.

Compliance with precisely speciﬁed time windows must be also assured, particularly for dark processing functions, i.e., business functions that usually process large volumes of data automatically during nighttime hours. For example, a bank’s transfer system must be able to post all transfers entered during the day overnight and before the start of the next working day. Even if this involves several million transactions, it must be ensured that the booking process is completed on time.

###### Load Tests (also: stress tests)

Like a performance test, the load test of a system also serves to determine the system’s behavior within its target environment. However, the explicit objective is to test this behavior under particularly high demands on the available resources. This can be done either by deliberately depriving the system of resources, or by artificially generating massive amounts of data or function accesses. Examples of resource deprivation include removing available work or hard disk space, reducing network bandwidth, or lowering the processing speed of the database system. Examples of load generation are the simultaneous logon and usage by extremely large numbers of users, the input of very large amounts of data, or the initiation of a vast number of transactions. Load tests are executed to observe how the system behaves in the event of an overload, whether this results in the loss or inconsistency of data, and whether the system re-stabilizes when resources are added. The observed behavior can often also be used to derive indicators that can be utilized for monitoring the applications in their subsequent operation.

Tool support is often vital for performance tests and also comes into play at the time of load tests, if not before. In particular, the simulation of a large number of simultaneous users cannot be carried out without the use of test tools. The setup of test tools must therefore also be taken into account when designing and planning load tests. Depending on the complexity of the application, the test tools themselves are smaller software projects.

###### Additional Test Types

The following test types for the execution of system tests are also identiﬁed (Balzert 1998, pp. 539–542):

* + - Security tests: running typical attack scenarios and ensuring that the system meets the required security attributes
    - Usability tests: methodical review of the usability of the system with regard to the speciﬁed usability requirements, which is carried out, for example, as an experiment with testing participants from a future user group
    - Restart tests: examining whether the system can be restored to its original operating state after a failure. This particularly includes importing backups and reviewing whether the original system state, including all application data, can be restored.

###### Regressability of System Tests

The term *regression tests* is used to describe test cases that have already been successfully run and are re-run after the system has been modified. The *regressability of test cases* therefore denotes the ability to repeat a test.

Systematic Testing of Software

In general, all test cases created should be capable of regression. This particularly applies to test cases within evolutionary software processes in which the software system is developed over the course of several iterations. For example, Figure 7 above shows a software process in which each iteration is concluded with a system test. This means that the entire system is tested at the end of each iteration. However, it is not just the newly added functions in each iteration that are tested: all the functions previously implemented are tested, too. This ensures that modifications to the program code do not inadvertently affect existing functions. However, even if the project does not use evolutionary development, the existing system test cases must be run for each new release of the software, even if the release cycles last several months or years.

The ability to continuously repeat all the existing test cases significantly contributes to achieving a continuously high level of quality. Therefore, when designing and executing test cases, emphasis should be placed on automating tests as much as possible. If this is not possible, the test cases must be designed and documented in such a way that they can be executed at any time, even by individuals who have not previously been involved in the project.

At present, various tools and frameworks are also available for the automation of GUI tests and are used in practice.

### Acceptance Tests

The final test level is the acceptance test. Here, the finished system is installed at the customer’s site and tested under actual operating conditions. The acceptance test examines whether the system meets the contractually agreed performance characteristics from the customer’s point of view. The customer personally carries out the acceptance test or is involved in its execution. In terms of the way it is executed, the acceptance test is a special type of system test, and it is generally difficult to draw a clear distinction between the two. The most important difference from system tests, however, is that it is executed by an organization other than the one that developed the system. A successful acceptance test is often the contractually agreed prerequisite for billing. In contrast to system tests, acceptance tests are typically conducted only once in the software process, as also shown in the example in Figure 7 above, whereas system tests are often conducted after each iteration or milestone.

###### Negotiation of the Test Result

While the previous test levels are generally carried out by the system manufacturer, the acceptance test is the responsibility of the customer who has commissioned the system. Therefore, the test cases for the acceptance test are directly oriented to the acceptance criteria agreed between the customer and the supplier.

In contrast to the other test levels, a decision is made at the end of the acceptance test: acceptance or rejection. In practice, it is very unlikely that no more errors will be identiﬁed during an acceptance test. Therefore, after an acceptance test has been executed, negotiations are usually held to determine which errors still need to be corrected for final acceptance. Acceptance is ofﬁcially granted only when these errors have been corrected.

Summary

Depending on the current progress of a project, software is tested at different test levels: component tests, integration tests, system tests, and acceptance tests. For the methodical execution of software tests, test requirement analysis, test planning, test case speciﬁcation, test data generation, test execution, and test evaluation activities must be conducted independent of the test level.

The isolated examination of individual software components is referred to as component tests. These tests are supported by automated unit tests and test-driven development. Integration tests examine whether the components work together as described in the speciﬁcation, using drivers and dummies as required by the integration strategy. The objective of system tests is to review whether the system as a whole meets the speciﬁed requirements. In addition to functional tests, performance tests and load tests are also executed. Acceptance tests examine whether the system meets the contractually agreed performance characteristics from the customer's point of view. A successful acceptance test is often the contractually agreed prerequisite for billing.



# Unit 7

## Systematic Quality Assurance of Requirements, Architectures, and Processes

#### STUDY GOALS

After completing this unit, students will know ...

... which activities are necessary for the quality assurance of requirements.

... when the quality of architectures can be ensured in the software process.

... which quality attributes and maturity levels can be determined for software processes and how these processes can be improved.

DL-D-IQSS01–L07

1. Systematic Quality Assurance of Requirements, Architectures, and Processes

### Introduction

Quality assurance activities in the software process often focus on the software system that has been created. However, the requirements and architecture definitions documented during the software process must also be examined. This unit presents suitable approaches for this. Alongside this, the quality of the software process can also be evaluated and improved.

### Quality Assurance of Requirements

The analytical quality assurance of engineering requirements is conducted in a core activity designed to examine and coordinate. The objective of these activities is to approve the requirements for implementation in the downstream phases of the software project. In the example process in Figure 7 above, analytical quality assurance of requirements is conducted after the business requirements have been determined and then after the technical speciﬁcation has been created. This is to ensure that the quality of the determined and documented requirements is assured before they are processed further. This is particularly necessary because all further activities in the software development process are directly or indirectly dependent on the documented requirements. Each error in these requirements that is not detected early will propagate through the technical speciﬁcation and architecture to the program code.

In addition, it is particularly important for business requirements to be coordinated with all relevant stakeholders. The set of requirements approved for implementation corresponds to the work order of a software project. In addition, once requirements have been implemented, they cannot be easily modified or removed from the system. Therefore, an established coordination process that accompanies the project helps with selecting the most important requirements to be implemented. In process models that support evolutionary development, for example, the requirements to be processed are defined anew for each cycle. Coordination between the stakeholders is required as soon as contradictions have been identiﬁed in the set of documented requirements. A conﬂict situation exists if the conflicting requirements originate from different stakeholders, each of whom also is pursuing conflicting objectives within the project. Another trigger for a conflict is the identiﬁcation of a contradiction between the documented requirements and the point of view held by a stakeholder. Reasons for this situation can be, for example, requirements for the system that have changed over the course of the project or incomplete requirements determination. In practice, it is apparent that conflicting requirements often result from a different understanding of the concepts, responsibilities, and motivation of the project. The activities for the examination and coordination of requirements can be divided into four steps:

Systematic Quality Assurance of Requirements, Architectures, and Processes

1. Determine examination criteria.
2. Select examination principles and examination techniques.
3. Conduct examination and document results.
4. Coordinate requirements/conflict management.

###### Determine Examination Criteria

The principle of *complete testing is not possible* also applies to the quality assurance of requirements. In practice, even requirements documents are usually neither completely error-free nor complete. In addition, industrial software development is a knowledge-driven process, and the set of business requirements typically changes several times during the course of a development project. For this reason, the criteria according to which the set of requirements is to be accurately examined must be determined prior to the examination. Which criteria these are in detail depends largely on the project situation, the type and importance of the requirements, the time available, and the people scheduled for this. The designated requirements engineer is responsible for creating the criteria.

Examination criteria for requirements can be assigned to three different quality aspects: content, documentation, and coordination.

Examine the Content

The central question to be answered by the criteria for the quality aspect content is: “Are all relevant requirements documented at the required level of detail so they can be understood?”

Examine the Documentation

The criteria for examining documentation are aimed at the use of appropriate forms of documentation and the quality of documentation. The criteria for examining documentation answer the central question: “Are they documented in an understandable manner and in compliance with the instructions for documentation?”

Examine the Criteria for Coordination

The criteria for examining the coordination of documentation are intended to

answer the question: “Do all relevant stakeholders agree with the documented requirements and have the known conﬂicts been resolved?” Particularly through gaining new knowledge during the software process, new requirements can be identiﬁed that contradict those already documented.

###### Select Examination Principles and Examination Techniques

The examination principles and examination techniques to be considered are only selected after the relevant examination criteria have been decided and when the amount of resources available for the examination has been determined.

Various examination principles can be distinguished within the scope of examination requirements that do not represent an examination technique in themselves, but help in the selection of examination techniques. In contrast, examination techniques are specific procedures or approaches that are used to execute the examination. An examination technique can support one or more examination principles.

The six examination principles described below are suitable for improving the quality of requirements examination results (cf. Pohl 2021, p. 106ff.):

* Principle 1: Involvement of the right stakeholders.
* Principle 2: Separation of error detection and error correction.
* Principle 3: Examination from different points of view.
* Principle 4: Appropriate change in the documentation form.
* Principle 5: Design of development artifacts.
* Principle 6: Re-examination.

Since requirements, unlike software artifacts, cannot be executed, only static quality assurance activities can be undertaken. The review techniques already presented are particularly relevant to the examination of requirements. In addition, however, requirements are also quality-assured through the tentative design of development artifacts or the transfer to another form of documentation.

###### Conduct Examination and Document Results

The examination must be organizationally planned after the business preparation is complete. This includes organizing the required infrastructure, scheduling all the participants and, if necessary, coordinating the external experts. It is important to concentrate on documenting the results of the examination. Troubleshooting and revision of the documented requirements occurs afterward and is not part of the examination.

###### Coordination of Requirements/Conflict Management

If conflicting requirements have been identiﬁed and cannot be easily resolved, a conﬂict situation exists. The activities for coordinating requirements are intended to achieve a common and consistent understanding of the set of requirements among the stakeholders (cf. Pohl 2021, p. 102). Alongside this, decisions must also be made about which requirements should actually be implemented. Conflict management is to be used to resolve the conflict while maintaining the stakeholders’ willingness to cooperate at the same time.

Systematic Quality Assurance of Requirements, Architectures, and Processes

### Quality Assurance of Architectures

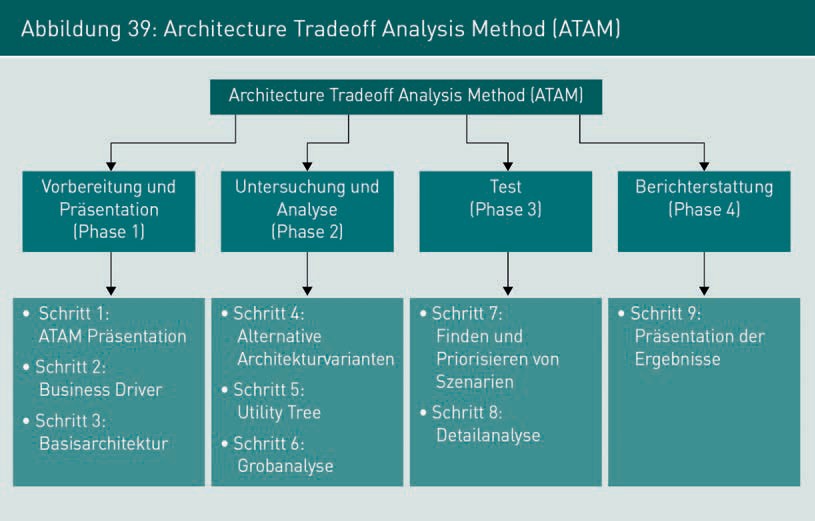
When and with which techniques the quality of architectures is examined and evaluated fundamentally depends on the objective of the examination: on the one hand, the suitability of a designed architecture can be evaluated before implementation (ex ante) and on the basis of the architecture description. On the other, the suitability of a designed architecture can be examined during implementation and after the end of implementation (ex post) to determine whether the architecture actually created adheres to the specifications of the architecture description created before implementation. These two examination objectives are so divergent that different examination techniques are used for each of these examinations.

###### Evaluation of Architectures before Implementation (ex ante)

The evaluation of an architecture before its implementation is intended to assert the suitability of an architecture for fulfilling the requirements of a system. This evaluation generally correlates with the fulfillment of quality attributes and constraints, since these types of requirements have the strongest influence on the choice of a suitable architecture. Techniques of scenario-based architectural analysis are applied for this and the architecture tradeoff analysis method (ATAM) is often used. The fundamental principle of scenario-based architecture analysis is the evaluation of architecture deﬁnitions based on specific application scenarios of the system to be created. After an architecture deﬁnition has been created and documented, it is investigated with the help of selected application scenarios. In the process, the architecture deﬁnition is evaluated to determine in particular whether, the quality attributes required by the application are supported.

###### ATAM Phases and Activities

The architecture tradeoff analysis method not only includes activities for evaluating architecture deﬁnitions, but also combines the evaluation with finding possible alternatives. Figure 39 below depicts the phases and activities of ATAM that are described in more detail in the following (see Starke 2020, pp. 310–317).



Phase 1: Preparation and Presentation

The first phase of ATAM serves to prepare and present the basic architecture with the involvement of all relevant stakeholders and, if necessary, also the users or technical experts from the specialist departments. In Step 1, the ATAM approach is presented to all stakeholders and the activities it consists of are explained. In Step 2, the most important functional requirements, quality requirements, and boundary conditions are presented. In ATAM, these are referred to as business drivers. Step 2 aims to ensure the objectives of the software system through a common understanding of all the stakeholders involved in ATAM. Step 3 of Phase 1 is the presentation of the basic architecture created by the architects. This is an architectural design that still needs to be refined, but already contains the fundamental architectural decisions. The objective of Step 3 is to present the current status of the architectural deﬁnition and clarify any points that may still be unclear.

Phase 2: Investigation and Analysis

Using the findings from the first phase, the architecture team draws on the basic architecture to create various architecture variants in Step 4 which can be used to fulfill the functions and attributes required of the system. In view of the evaluation of the architecture in Phase 3, application scenarios are created and prioritized in Step 5, where the specifically required quality attributes are described. The set of scenarios with quality attributes is referred to as the utility tree in ATAM. In Step 6 of ATAM, the architecture variants created in Step 4 are roughly analyzed for fulfillment of the scenarios derived in Step 5. This involves analyzing the extent to which the different architecture variants support the set of quality attributes in the utility tree. This rough analysis results in a prioritized list of architecture variants that is used as input for Phase 3 in ATAM.

Systematic Quality Assurance of Requirements, Architectures, and Processes

Phase 3: Testing

The objective of Phase 3 is to determine the most suitable architecture variant. To this end, the set of scenarios created in Step 5 is first refined, expanded, and prioritized in Step 7. All the relevant stakeholders are taken into account. In Step 8, a detailed analysis of the architecture variants is then carried out using the set of scenarios detailed in Step 7. This testing is similar to Step 6, but is now based on the expanded and more detailed scenarios. In the process, the individual architectural variants are determined, and their suitability is explored. An overall assessment is also prepared, documenting the dependencies between individual quality attributes and the degree to which the attributes are achieved using the various architecture variants, as well as the associated risks. In general, tradeoffs must be made in the final architectural decision with regard to the achievement of quality objectives. The objective of the detailed analysis is to work out these tradeoffs.

Phase 4: Reporting

The objective of Phase 4 is to document the results of the analysis and the architecture decisions made. The conclusion of ATAM is the reporting to the stakeholders involved, which takes place in Step 9. Here, the knowledge gained over the course of Steps 1 to 8 is summarized with regard to the required quality attributes and the architecture decisions based on them. When documenting architectural decisions, particular attention must be paid to their traceability. If design decisions are made, the bases of decisions already made must be traceable in the further course of the software process.

###### Evaluation of Architectures during/after Implementation (ex post)

The purpose of examining the architecture during implementation and/or after implementation has been completed is to ensure that the result of the development work adheres to the specifications established by the architectural deﬁnition. These activities can be compared to construction supervision on a building site: the individual trades must be coordinated and their intermediate and final results checked to ensure that they comply with the design plan in terms of type and quality. When creating a software system, the architect must also ensure that the decisions made by the architecture team have actually been implemented in the program code. One technique for this is what is known as the **architecture compliance check**. The basic idea is to create a software model based on the actual program code of the application, for example, in the form of a UML class diagram or UML component diagram. This software model is compared with the model used to describe the architecture before implementation. If deviations can be identified between the two models, they must be evaluated by the architect and remedied by the development team, depending on the evaluation.

Architecture Compliance

Check

This is an approach to architecture evaluation in which the software models of the architecture deﬁnition are compared with the models generated from the implemented code.

Note

Regular and structured quality assurance of architectures is often not standard in industrial practice and is used only sporadically, if at all. There is often a lack of awareness of the necessity of these activities. Moreover, if they are not considered from the outset when planning the project, the necessary resources will not be available for this during project implementation.

### Quality Assurance of Software Processes

In addition to the artifacts created in a software process, the quality of software processes or software process models can also be examined and improved. The basic assumption behind the quality assurance and improvement of software processes is that there is a relationship between process quality and product quality. Various speciﬁc process characteristics can be used to evaluate software processes. If a need for change is identified, this can be implemented as a process change procedure.

###### Quality Attributes for Software Processes

In order to identify the need for change in software processes, a process analysis must first be carried out. This can be done quite pragmatically by surveying the stakeholders involved in the process. Another option is the structured evaluation of various quality criteria, for example, by the stakeholders in the software process. Figure 40 below shows 10 different characteristics that can be evaluated for processes.

|  |  |
| --- | --- |
| Figure 40: Software Process Characteristics | |
| Process Characteristic | Description |
| Understandability | To what extent is the process explicitly deﬁned and how understandable is this deﬁnition? |
| Standardization | How much is the process based on a generic standard process? This may be of interest to some customers who rely on consistency with a set of deﬁned process standards. To what extent is the same process used in all areas of a company? |

Systematic Quality Assurance of Requirements, Architectures, and Processes

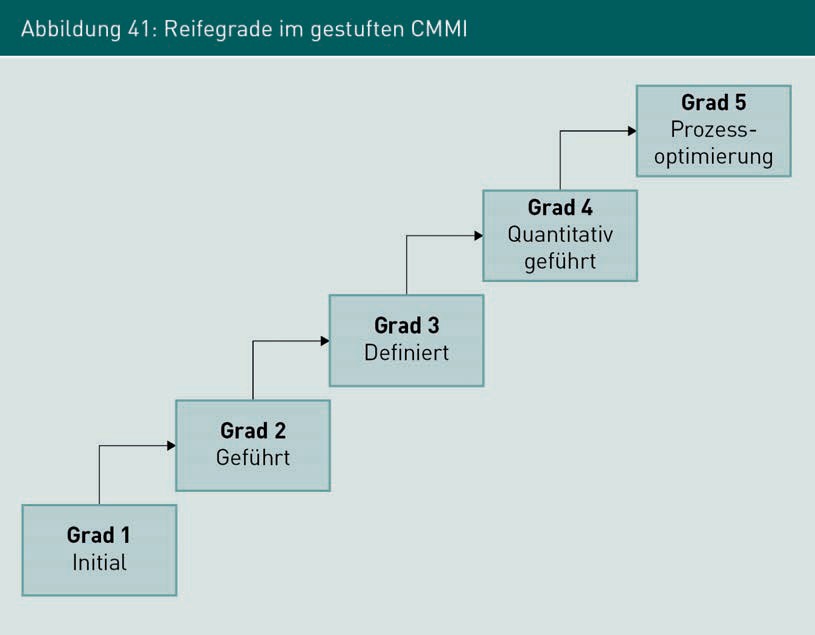
|  |  |
| --- | --- |
| Process Characteristic | Description |
| Recognizability | Do the process activities end with clear results, so that the process progress is externally recognizable? |
| Measurability | Does the process include data collection or other activities that allow the measurement of process or product attributes? |
| Supportability | To what extent can software tools be used to support process activities? |
| Acceptance | Is the deﬁned process acceptable to and useable bythe developers of the software product? |
| Reliability | Is the process designed to prevent or detect process errors before they cause product errors? |
| Stability | Can the process continue despite unforeseen problems? |
| Ease of maintenance | Can the process be further developed to reflect changed operational needs or process improvements? |
| Speed | How quickly can the process of delivering a system from a provided speciﬁcation be completed? |

###### Capability Maturity Model Integration (CMMI) Maturity Model

The American Software Engineering Institute (SEI) developed a maturity model for software processes that is referred to as capability maturity model integration (CMMI). The CMMI is designed as a framework for process improvements. Among other things, it contains a staged model for assessing the maturity of software and management processes. The CMMI is highly comprehensive and includes:

* 22 process areas from software engineering,
* objectives that are formulated speciﬁcally for each process area, and
* procedures by which the objectives can be achieved.

The current capability level of a process is determined on a 5–stage scale. The determination of the maturity level is based on the analysis of the current ACTUAL processes of a company. CMMI Level 1 is the lowest level and Level 5 corresponds to the highest capability level. Figure 41 below shows the staged CMMI maturity model.



The following five stages can be identified in the CMMI model:

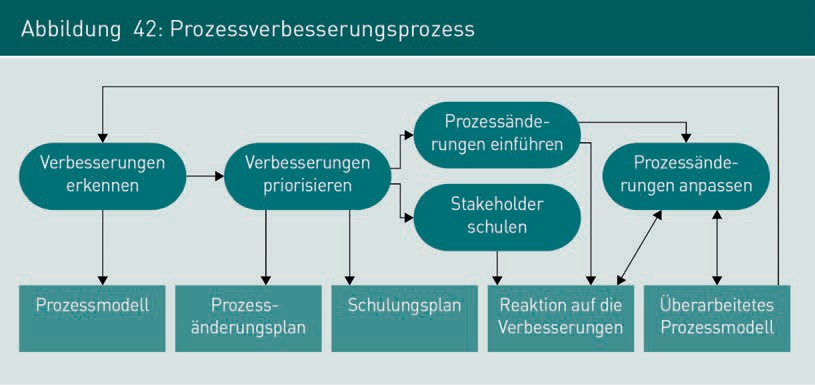
* + Stage 1—Initial: there is no process deﬁnition; activities ﬁrst take place *ad hoc* or are performed chaotically. The process is usually not repeatable.
  + Stage 2—Managed: the process is repeatable. The scope and outcome of the activities to be completed is known to the team members. However, there is no detailed process that explicitly deﬁnes activities, roles, results, and their dependencies.
  + Stage 3—Deﬁned: there is a deﬁned process in which activities, roles, and results are documented in a binding manner. The process contains specific methods and approaches for the individual software engineering activities.
  + Stage 4—Quantitatively Managed: process quality can be determined and managed by measuring quantitative process characteristics. By collecting key figures on the activities of the processes and on the results generated, problems can be identified and counteracted accordingly.
  + Stage 5—Process Optimization: process and product measurements are continuously collected and used for process improvement. Depending on the value measured, the process can be adapted to current requirements.

Systematic Quality Assurance of Requirements, Architectures, and Processes

According to the CMMI model, the maturity of an organization should gradually develop from a lower to a higher level. CMMI maturity levels can therefore be used to evaluate an organization. In addition to the staged CMMI model, there is also a continuous CMMI model. In this case, the maturity level is not determined for an entire organization, but rather for various process groups.

###### Procedure for Process Improvement

In the event that suggestions for improvement were developed during the process analysis, these must be transferred to the software process model in a structured manner. For this purpose, a procedure for process improvements can be established and is shown as an example in Figure 42 below. This procedure provides an overview of the typical activities to be conducted during a process improvement and their results. The procedure is cyclical, which essentially means that it will never be final. Even an improved process model can be further improved over time, which corresponds to the implementation of the principle of continuous improvement as stated in TQM.



The first activity in the improvement process is the identification of areas in need of improvement in the current process version and the development of possible proposals for making the necessary improvements. The proposals are then prioritized and their feasibility analyzed, along with the further activities required for implementation. Prioritizing in this way results in a plan for changing the process, which documents, among other things, the timing and responsibilities for the introduction of the improvements, as well as potential interactions between the individual improvement steps.

To facilitate this, a training plan is drawn up, on the basis of which specific training measures are introduced for the stakeholders affected by the changes. In addition to knowledge of the changes in the software process, new technical competencies may also need to be acquired in order to implement the process changes. This is ensured by training the affected stakeholders. The introduction of process changes often means that templates, documentation, tools, and procedures must be adapted, and the adaptations observed and evaluated. Particularly in the case of major changes, it is often necessary for their specific implementation to also require minor adjustments at various points. Therefore, in-depth discussions between the affected stakeholders and the process designers should occur during the adaptation phase. Once the adaptation phase has been completed, the software process model has again reached a stable state and the improvement cycle can start anew.

Summary

Analytical quality assurance of requirements is conducted in a core requirements engineering activity, designed to examine and coordinate. The objective of the activities for examining and coordinating is to approve the requirements for implementation in the downstream phases of the software project. The quality of architectures can be evaluated before implementation (ex ante) and on the basis of the architecture description. A technique frequently used for this purpose is scenario-based architecture analysis, for example, with ATAM. The fundamental principle here is the evaluation of architecture deﬁnitions based on specific application scenarios of the system to be created. Alongside this, the architecture can be examined after the completion of implementation (ex post) to determine whether the architecture actually created also meets the specifications of the architecture description created before implementation. Various speciﬁc process characteristics can be used to evaluate software processes or to determine their maturity. If a process is to be adapted, this can be implemented with a process change procedure.



# Appendix 1

## List of References

134 Appendix 1

List of References

Balzert, H. (2008): *Lehrbuch der Software-Technik: Software-Management.* [*Textbook of Software Engineering: Software Management].* 2. Auﬂage, Sprektrum Akademischer Verlag, Heidelberg.

Balzert, H. (1998): *Lehrbuch der Software-Technik, Software-Management, Software- Qualitätssicherung, Unternehmensmodellierung.* [*Textbook of Software Engineering, Software Management, Software Quality Assurance, Company Modeling*].Spektrum Akademischer Verlag, Heidelberg.

Checkstyle (o. J.): (URL: https://checkstyle.sourceforge.io/checks.html [letzter Zugriff 28.02.2022])

SpotBugs (o. J.): *Bug Descriptions.* (URL: https://spotbugs.readthedocs.io/en/latest/ bugDescriptions.html [letzter Zugriff 28.02.2022])

Gabler Wirtschaftslexikon (2014): Springer Gabler Verlag. (URL: https://wirtschaftslexi- kon. gabler. de/deﬁnition/qualitaetssicherung-44396/version-267707 [Definition: What is “Quality Assurance”]. [letzter Zugriff 28.02.2022])

German Testing Board (2014): S. 43. (URL: [https://www.german-testing-board.info/wp- content/uploads/2022/01/GTB-CTFL\_Lehrplan\_v3.1\_DE.pdf](https://www.german-testing-board.info/wp-%20content/uploads/2022/01/GTB-CTFL_Lehrplan_v3.1_DE.pdf) [Syllabus Certified Tester Foundation Level]. [letzter Zugriff 28.02.2022])

Grechenig, T. et al. (2010): *Softwaretechnik. [Software Engineering].* Addison-Wesley Verlag, Munich.

Kleiner, F. (2021): *IT Service Management.* mitp Verlag, Frechen.

Liggesmeyer, P. (2002): *Software Qualität – Testen, Analysieren und Veriﬁzieren von Soft- ware.* [*Software Quality – Testing, Analyzing, and Verifying of Software*].Spektrum Verlag, Heidelberg.

Ohno, T. (2006) *“Ask 'why' ﬁve times about every matter.”* (URL: http://www.toyota-glo- bal.com/company/toyota\_traditions/quality/mar\_apr\_2006.html [letzter Zugriff: 04.08.2014])

PMD (o. J.): *Current Rulesets.* (URL: <http://pmd.sourceforge.net/pmd-5.1.1/rules/> index.html [letzter Zugriff: 28.02.2022])

Pohl, K./Rupp, C. (2021): *Basiswissen Requirements Engineering - Aus- und Weiterbil- dung zum “Certiﬁed Professional for Requirements Engineering”.* [*Basic Knowledge of Requirements Engineering—Training and Continuing Education to be a “Certiﬁed Professional for Requirements Engineering”*]. 5.Auﬂage, dpunkt.verlag, Heidelberg.

Schatten, A. et al. (2010): *Best Practice Software-Engineering.* Spektrum Akademischer Verlag, Heidelberg.

Schiersmann, C./Thiel, H.-U. (2000): *Projektmanagement als organisationales Lernen.* [*Project Management as Organizational Learning*]. Leske + Budrich Verlag, Opladen.

##### Appendix 1

135

List of References

Sommerville, I. (2018): *Software Engineering.* 10. Auﬂage. Addison-Wesley Verlag, Munich.

Sneed, H./Baumgartner, M./Seidl, R. (2012): *Der Systemtest.* [*The System Test*].Carl Hanser Verlag, Munich.

Starke, G. (2020): *Effektive Softwarearchitekturen*. *[Effective Software Architectures].* 9. Auﬂage. Hanser Verlag.

Wallmüller E. (1990): *Software-Qualitätssicherung in der Praxis.* [*Software Quality Assurance in Practice*]. Carl Hanser Verlag, Munich.