SpringerBriefs in Information Systems

Improving Software Testing

Technical and Organizational Developments

Bearbeitet von
Tim A. Majchrzak

1. Auflage 2012. Taschenbuch. XVIII, 160 S. Paperback
ISBN 978 3 642 27463 3
Format (B x L): 15,5 x 23,5 cm
Gewicht: 284 g

Chapter 2
Software Testing

Software testing as a part of software development is a very diverse topic—or even seen as an *art* [1]. Therefore, an introduction into software testing is given. This chapter summarizes why software is tested, which terms are important, what software testing is, how software testing can be done, and how it is organized. Furthermore, basics of test automatization, management, and documentation are described. Additionally, testing standards are mentioned.

The structure of the chapter is aligned with standard literature on software testing [2–7]. In the literature there is no common order in which topics are introduced but the intersection of various sources helps to arrange an adequate synopsis. While the chapter draws a holistic picture of testing, aspects that are important for the topics of this book are given special attention.

Besides giving the background for the following chapters, this chapter can be used as a brief introduction to software testing and as a reference for testing techniques.

2.1 Testing Basics

First of all, the term *defect* is introduced. Then, the aims of testing can be sketched. Finally, a classification for testing is proposed and important standards are named.

2.1.1 Defects

A variety of terms is used to describe the fact that software is flawed. The seemingly arbitrary or even synonymous usage of terms such as *bug*, *defect*, *error*, *failure*, *fault*, *mistake*, or *problem* is confusing.[1] Not all of these terms are (directly) related

[1] Switching to German does not help. There are less terms commonly used, but *Fehler* can be used as a translation for all of the above mentioned terms besides *mistake*.

T. A. Majchrzak, *Improving Software Testing*, SpringerBriefs in Information Systems,
DOI: 10.1007/978-3-642-27464-0_2, © The Author(s) 2012

to software testing, and some of them even have contradicting definitions in the literature. Moreover, it has to be discussed what the actual meaning of terms is and which kind of states of a program or components of it have to be distinguished in the domain of software testing. It e.g. can be asked whether software is flawed if it does not work *at all*, does not work as *intended*, or does not work as *specified*.

Terms are not formally defined but their meaning with regard to software testing is presented. If possible, a consensus of explanations found in the most prevalent sources is given. All terms introduced in this section are also listed in the glossary in the backmost part of this work (p. 137ff.). Since it is the most important term, *defect* is introduced first and the other terms' relations to it are explained.

According to Cleff, there are two definitions for *defects* [5, p. 10ff.]:

- The *property based definition* addresses the absence of a guaranteed property of a software. Usually, properties are based on the specification and have been contracted between principal and agent. A defect in this sense is characterized by not fulfilling a specified performance [6, p. 28f.]. It has to be distinguished from a case in which software does not perform as expected. A property based defect is statically bound to the program code [2, p. 5ff.]—this does not mean that it will deterministically occur, though.
- The *risk based definition* can be derived from product liability judicature. It addresses risks that arise from defective software. Defects could potentially cause harm and pose the unauthorized or misappropriate usage of software.

Defects may also be called *faults*; in this work, the term *defect* will be used. It has been suggested to include occurrences in which software does not behave as intended by its users as defects [6, p. 28f.]. While software should be intuitive in order to avoid wrong usage and make it ergonomic, this definition is problematic. Software that does not behave as expected might be deficient in terms of jurisdiction (w.r.t. product liability), and requirements engineering (RE) could have failed, but including such cases as defects is prone to arbitrariness. Testing has to be done based on a specification or following general norms and best practices. This could include *usability* checks if usability is part of the specification [8]. However, general assessment of a program's usability is concerned by the fields of *usability engineering* [9, 10] and human-computer interaction (HCI) [11–13].

The term *bug* can be used to subsume defects and unexpected behavior of programs which do not violate the specification. Therefore, it can be used in a rather informal way to describe any characteristic of a program that is undesired from a user's view [14, p. 19f.]. Bugs thus can be signs of defects. However, one bug might be caused by multiple defects or other way around.

Terms that can be defined more precisely are *failure* and *error*. Errors describe a mistake in the life cycle of a software that leads to an incorrect result [3, p. 354ff.]. They (at least indirectly) are caused by mistakes humans make or their misunderstanding of a program's specification, the programming language used, or other vital parts of the development process [2, p. 7]. If the consequence is identified when

executing a program, a failure has been found; this failure hints to the existence of a defect.

While a defect is static, failures occur *dynamically* [2, p. 6f.]. Even though some defects will cause failures regardless of the way a program is used, failures can only be identified when executing a program and are often bound to the input given to it.

The relationship of error, defect and failure can be illustrated with an example (cf. [3, p. 354f.]). A simple program expects two parameters a and b and prints the sum of a and two times b. Consider the main statement of the program to be[2]:

```
print a + b * 2;
```

If the programmer typed too fast or did not work carefully, he could type a wrong number. In other words: an error occurred. The actual statement would then e.g. be:

```
print a + b * 3;
```

This is the defect. If the program is tested, a failure might be observed. If the tester knows that the program was specified to calculate $a + 2b$ and he tested it with parameters $(2, 5)$, he would notice that the output was 17 instead of 12 as expected. Ideally, this would lead to the discovery of the defect. At the same time it becomes clear that testing does not necessarily reveal defects. If the (only) input would have been $(2, 0)$, there would not have been a failure—despite the defect.

When testing software, each condition that violates a program's specification is called an *incident* [5, p. 10]. Incidents can be failures. However, their cause can be external. A malfunctioning testing environment, an erroneous test tool, mistakes while performing the test, or insufficient user rights of the tester might be the reason for a condition that would also be expected if the tested program contained defects [5, p. 10]. Only failures should be reported to software developers in order to fix them [5, p. 10] because external effects are beyond their responsibility. Reporting them should include documenting them, preferably following a standard such as IEEE 1044 [15].

It is suggested to classify defects as well as failures. From a tester's perspective, even each incident should be categorized in order to decide how to proceed with it. Typically, classification is done by specifying a level of *severity*, a *priority*, incident related information, and optionally comments [14, p. 34f.].

In general, no other terms to describe the cause or signs of malfunction of software will be used in this work. Additionally, the term *test case* has to be defined. A test case is the description of an atomic test. It consists of input parameters, execution conditions, and the expected result [4, p. 171]. Execution conditions are required to set up the program for testing, e.g. if the test case requires it to be in a defined state. Test cases not necessarily are literal descriptions of tests; they also can be on hand as executable code or in a format specified by a test tool. Test cases that are stored in

[2] No actual programming language is used. A file header and a function or method definition have been omitted. The statement would look similar in many common object orientation programming languages, though. Some languages would require a *cast* from the numeric type to a literal type or a parameter for formatting.

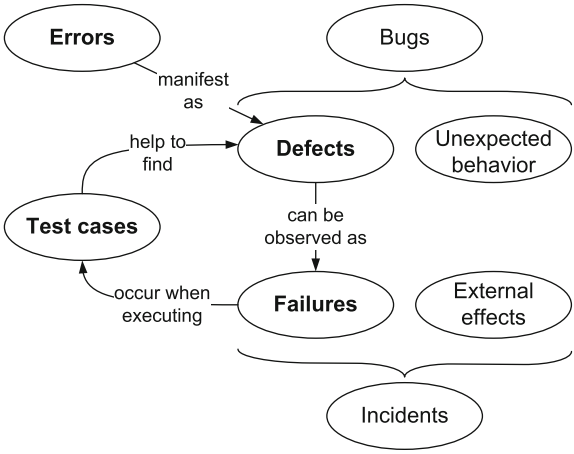

Fig. 2.1 Testing terms and their relations

executable form, i.e. that are small programs by themselves, can also be called *test drivers* [16, p. 132].

The most important terms and their relations are summarized in Fig. 2.1. Of course, there are many more terms in the domain of software testing. Some of them are explained in the following sections; many more are explained in the glossary.

2.1.2 Aims of Testing

"To test a program is to try to make it fail" [17]. Thus, finding failures is the main aim of testing [5, Chap. 3, p. 59f.]. This aim aligns with the question *Why test software?* asked at the beginning of this book (Sect. 1.2). Despite its charming simplicity, subordinate aims can be identified in definitions on software testing. Testing *without* aims can be seen as a waste of time and money [18, p. 223].

According to the IEEE Standard 610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology)[3] testing can be defined as follows [19, 20]:

1. "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."
2. "The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items."

[3] The standard lists further definitions for testing. This includes acceptance testing, component testing, functional testing, integration testing, loopback testing, mutation testing, performance testing, regression testing, stress testing, structural testing, system testing, and unit testing.

The first definition suggests that finding failures can be accompanied with an evaluation. This, for example, applies to testing a program's performance. Strictly speaking, identifying an unsatisfactory performance also is a failure since the system does not perform as specified. Thinking of performance testing when trying to find failures is not obvious, though. The second definition is somewhat fuzzy but aligns with the above mentioned considerations.

Omitting superfluous arguments and aims that are almost equal to finding failures, literature hardly mentions subordinate aims. In fact, many authors do not explicitly define the aims of testing at all but either take them as given (for instance [6]) or embed testing in a context of organizational optimization (e.g. [21]). Other authors describe aims of tasks that testing is embedded into (such as *quality management* [18]), or aims derived from roles in the testing process (for instance *test analyst* [3]). Furthermore, it is common to discuss aims for quality which should be reached using software testing [2, 4]. These aims include security, applicability, continuity, controllability, functionality, usability, integrability, performance, and efficiency [22, Chap. 12].

If subordinate aims are discussed, they are found in the context of test execution rather than in the context of motivating why testing is pursued. Typically, these aims are general aims of software development; software testing can be used to reach them or to close the gap between the current status and fulfilling the objectives. Such aims are

- decreasing costs to remove defects,
- decreasing costs defects cause (due to system failure),
- assessing software quality,
- meeting norms and laws,
- avoiding lawsuits and customer related problems,
- increasing the trust into the software developed or into a company that develops software products [5, p. 59ff.], and
- supporting *debugging* as good as possible [14, p. 54f.].

These aims have an economic background. They either try to decrease costs or increase revenue both directly and with keeping long-term effects (such as a company's image) in mind.

From the testing perspective, all aims can be subsumed to finding failures. However, it can be argued that the aim of testing is to increase the value of software. Eliminating defects (and, to be able to do so, finding failures) is the instrument to reach higher quality and, therefore, value [23]. Increasing value has to be kept in relation with testing effort to keep the aim operational.

Besides the aims of testing from the external perspective, testing as seen as a task in developing software also should have aims. These aims should be "objectively measurable" [18, p. 245] and usually try to achieve quantitative criteria such as coverage or a modification of complexity figures.

2.1.3 Classification and Categorization

Due to its diversity, software testing has a variety of facets. A basic categorization was carried out by structuring this chapter. In addition, testing can be ordered by *phases*. While testing is part of an overall development process, it can be split into several subprocesses. Test processes ought to strive for quality improvements [4, Chap. 20]. A framework of testing processes is explained in Sect. 2.3.1.

Testing techniques—also referred to as *methods*—can be classified by their characteristics. Hence, possibilities for classification are presented which will be used for the methods introduced in Sect. 2.2.

Common classification schemes are by main characteristic of a test (cf. [2, p. 37f.]), by the target of testing (e.g. GUI or integration, cf. [5]), and by the available information (see below). It is also possible to resign from using a classification of techniques in favor of defining adequate techniques for each phase of testing [6, Chaps. 6–17]. For this book, a scheme based on main characteristics is used since it poses least ambiguity—especially in combination with splitting the test process into phases.

There are three main classes of testing techniques: *static* and *dynamic* techniques [2, p. 37ff.], as well as complete enumeration (*exhaustive testing*). The latter will not be considered further because it almost is without relevance and is only possible for the most simple programs [5, p. 54] [18, p. 224]. Consider a tiny program that takes two integer numbers as parameters and returns their sum. Assuming that the underlying functionality (basic I/O) has been *proven* correct, there are still $2^{32} * 2^{32} = 2^{64}$ possible combinations of the input parameters for 32 bit data types—for 64 bit integers it would even be 2^{128} ($\approx 3.4 * 10^{38}$) possibilities. A (structure oriented) dynamic test would show that one test case is enough.

Static testing does not create test cases since there is no execution of the program to be tested [2, p. 43ff.]. Static tests can be subdivided into techniques for *verification* and *analysis*. Verification is done *formally* or *symbolically*. It theoretically is possible to prove that a program behaves in the way it is intended to. The effort to do so is immense and static testing usually is done for small modules of programs in areas where correctness is crucial (such as the central control of a nuclear power plant). *Static code analysis* is a technique that checks the source code or some form of intermediary code (for instance *bytecode*) for the assertion of properties. Techniques are *style analysis*, *slicing*, *control flow analysis*, *data flow (anomaly) analysis*, and *code reviews* [2, Chap. 8]. Furthermore, the calculation of *metrics* allows to draw quantitative conclusions about the status of development and testing [24, p. 283ff.].

The main categorization of *dynamic techniques* is to divide techniques by their orientation on a program's *structure* or *functions*. Over 50 different dynamic techniques are known [25, p. 328], but only some of them are widely used. Structure oriented tests can be subdivided into techniques based on the *control flow* and based on the *data flow*. Structure based techniques assess a program's behavior by monitoring how it is executed, and how data is processed. This is done on a low technical level. Functional tests abstract from the low level execution but create test cases based

on the functionality a program should provide. Since the functionality is specified, they can also be called *specification oriented* tests. [3, p. 31].

Besides these two categories, four further classes of dynamic techniques can be identified. *Dynamic analysis* is used to detect *memory leaks* and *performance bottlenecks* [3, p. 128ff.], and for dynamic symbolic execution. Symbolic execution takes a special role. It can be seen a static verification technique and even be used to draw complete conclusions [2, p. 44f.]. However, it may also be combined with dynamic testing and thereby becomes an approach for creating test cases. The tool Muggl, which will be introduced in Chap. 4, uses this technique. Due to its character, symbolic execution can be seen as a technique of dynamic analysis.

In contrast to Liggesmeyer [2], *performance tests*, such as *load tests* and *stress tests*, are categorized as dynamic tests, too. They require the execution of a program but due to their focus on a program's performance can be seen as an own category. Defining them as other dynamic (e.g. functional) testing techniques being constantly repeated would be ambiguous and would neglect their importance.

Experience based tests can be subdivided into *intuitive testing* (*error guessing* [2, Chap. 5]), and *explorative testing* [3, p. 108f.]. Both techniques are completely manual. They are based on a tester's skill and discretion.

Finally, there is a number of further techniques which do not fall into the above described categories: *back-to-back tests* [25, p. 376], *mutation testing* [4, p. 548], *statistic testing* [25, p. 368], *random testing* [2, p. 208], and *using original data* for testing [26, p. 97].

The categorization proposed in this section is summed up in Fig. 2.2. It has to be noted that this categorization may be inadequate if techniques are categorized from a business angle. *Integration*, *system*, *acceptance*, *beta*, and likewise tests usually are more or less manually conducted dynamic tests. From a business perspective, it is more important that an acceptance test is done in cooperation with customers than which particular technique is used. However, for categorization purposes the above suggestion is adequate.

In many situations it also helpful to distinguish test techniques based on the information available when creating test cases. Dynamic tests can be classified as black-box tests, glass-box tests (also known as white-box tests), and gray-box tests. This classification is very useful when a less granular categorization than sketched above is needed. Such a lack of granularity can be criticized, though [2, p. 40]. Black-Box tests abstract from the source code. Most functional tests are Black-Box tests. Glass-Box tests utilize knowledge of the code; control flow oriented and data flow oriented techniques are Glass-Box tests. Gray-Box tests fall in between; they do not focus on utilizing the source code but are at least conducted by persons that have knowledge of it [7, p. 119].

A completely different method to classify testing techniques is proposed by Pezzè and Young. They suggest to define basic principles of testing which techniques should follow. These principles are *sensitivity*, *redundancy*, *restriction*, *partitioning*, *visibility*, and *feedback characteristic*. With regard to the chapters in their book, Pezzè and Young however follow a common classification scheme [4].

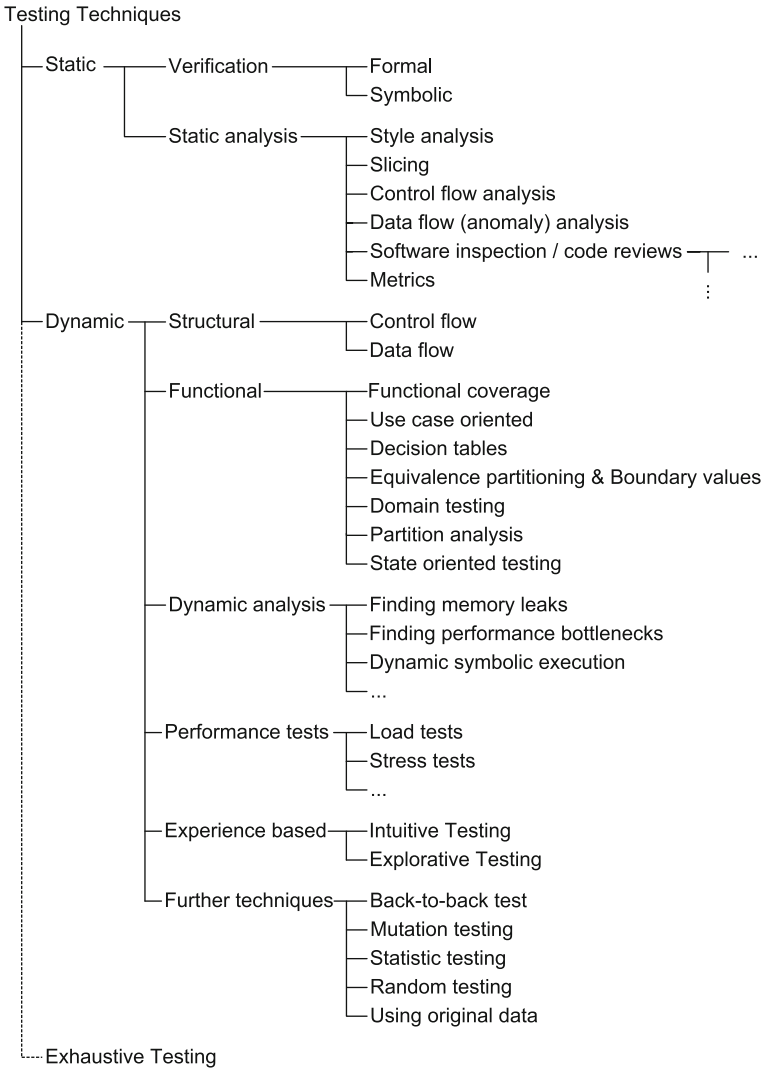

Fig. 2.2 Categorization of testing techniques (inspired by [2, p. 38])

It was decided to exclude *model checking* and *model based testing* from the categorization scheme. Model checking has gained much popularity in the last years, but despite being a verification technique, it usually is not seen as a testing technique. Model based testing is either based on model checking or subsumed by other techniques mentioned above. Examples for such techniques are the analysis of control flow and data flow or combinatorial techniques [4, p. 277]. Please refer to [27–29] for an introduction into model checking and to [30] for a survey on recent progress. Pezzè and Young introduce model-based techniques [4, Chap. 14]. It can be noted that

some techniques are similar to testing techniques introduced in Sect. 2.2. However, as the name suggests, model checking deals with *checking* a model with regard to the assertion of properties, and not with *testing* it.

2.1.4 Standards

As a consequence of the business criticality of testing and the usage of software in fields where failure endangers humans, standards have been developed. They describe what testing is, how subdisciplines of it ought to be organized, or which testing aims have to be met for distinct purposes of using software. This book roughly follows the categorization proposed by Spillner et al. [24, Chap. 13].

An elaborate discussion of standards and their application is out of scope of this work. A comprehensive introduction is given by Spillner et al. [24, Chap. 13]. They also name a myriad of standards that apply to different phases of testing and in the areas of processes, products, documentation, and methods. A list of applicable standards is also given in [18, p. 426ff.].

2.1.4.1 Terminology and Positioning

First of all, there are general standards that define terminology such as IEEE 610.12-1990 [19, 20] already mentioned in Sect. 2.1.2. These standards are complemented by more specific ones, for instance the ISO/IEC 12119:1994 Standard on *Information technology—Software packages—Quality requirements and testing* [31].

Besides these very general standards, testing organizations try to standardize the body of knowledge on software testing. Organizations such as the ISTQB [32] or their national associations such as the German Testing Board (GTB) [33] publish curricula. There are both general curricula and curricula that address advanced testing management or technical roles of testing. Commonly, curricula are available free of charge (e.g. from the GTB [34]) while literature that delivers the actual knowledge or is used to prepare for certifications (cf. Sect. 2.3.5) is published in books (such as [3, 24, 35]).

2.1.4.2 De facto Standards and Technical Specifications

A number of documents and best practices became de facto standards for testing without being published or without being denominated a standard. In particular, technical specifications can be seen as preliminary stages of standards [24, p. 360f.].

Technical standards are released by large software vendors and by consortia (both industry based and non-profit consortia). Typical examples are the standards for Web development released by the World Wide Web Consortium (W3C) [36]. Some technical standards specify aims for testing or quality criteria.

So called *bodies of knowledge*, which sum up best practices from a particular field, can also be regarded as de facto standards. An examples is the Software Engineering Body of Knowledge (SWEBOK) [37], which also comprises a chapter on software testing. Process models that evolved to be used in many projects also take a role as de facto standards. For example, the *V-Model XT* [38, 39] evolved from the V-Model 97 [40] and is widely used in development projects led by the German Government or German authorities. It also contains testing-related advices.

2.1.4.3 Company, Sector, or Application Related Standards

International enterprises, in particular multi-national software development companies, employ norms and standards for their development processes [24, p. 359f.]. In general, they are not officially published but used internally and in collaboration with key customers. They are especially used to provide templates, establish general terms and procedures, and to align processes.

Sector specific standards define requirements that have to be met in order to ensure the fail-safe or failure-tolerant operation of systems. Typically, requirements are very special or software is similar in design (e.g. it has to do a high number of physical calculations) [24, p. 361f.]. This applies to areas such as astronautics, aviation, healthcare, infrastructure, military, and telecommunication [24, p. 361]. The aviation norm *DO-178B* [41] (*Software Considerations in Airborne Systems and Equipment Certification*) defines testing techniques to be used and testing aims to be met with regard to aviation systems' levels of criticality. A similar norm is DIN EN 50128 (*Railway applications. Communications, signalling and processing systems*) which applies to railroad transportation systems [42]. For the healthcare sector, IEC 62304 regulates properties medical device software has to fulfill [43]. Testing can be used to check whether the requirements defined by standards are met [5, p. 71f.].

Application related standards address requirements of software that is tailored to a distinct domain of usage. This e.g. applies to *embedded systems* which often are used in sensitive areas. For instance, embedded systems control the anti-lock braking system (ABS) and the electronic stability control (ESC) that increase the safety of cars. While company and sector related standards are likely to apply in distinct fields, there are some standards that apply to application domains. In the above case, IEC 61508 (*Functional safety of electrical/electronic/programmable electronic safety-related systems*) [44] applies. It defines levels of security and boundaries for the expected probability of breakdowns.

2.1.4.4 General Standards

Spillner et al. [24, p. 365] identified almost 90 International Organization for Standardization (ISO) standards that concern software development and over 40 IEEE documents on software engineering. Without focusing on software testing directly,

applying these standards influences the way software is tested. They comprise terminology guidelines, process suggestions, documentation templates, and method advise [24, p. 365]. Thus, they especially regard the organizational implementation of testing and the interfaces to other software development processes.

Further standards that influence software testing are IT frameworks such as the IT infrastructure library (ITIL) [45] or Control Objectives for Information and Related Technology (CobiT) [46]. Some of them directly propose guidelines for testing (for instance CobiT) while others change the framing conditions for software development in general (for instance ITIL). Despite different aims, reference models and evaluation frameworks such as Capability Maturity Model Integration (CMMI) [47] influence software development and thereby testing in a similar way as IT frameworks.[4]

Testing related process models are no *de jure* standards but might be standards in companies. The most notable process models are Test Management Approach (TMap) [50, 51] and Test Process Improvement (TPI) [52, 53]. A detailed introduction is e.g. given by Pol et al. [22, Chap. 8].

Finally, even more general standards have relevance for software testing. This considers all standards and norms that deal with product quality, product and process security, liability, and other product or production related details. The most prominent example is a Total Quality Management (TQM) approach based on the DIN EN ISO 9000 [54], DIN EN ISO 9001 [55], and DIN EN ISO 9004 [56] norms. TQM is an continuous improvement process that aims at improving product quality and at involving the customer [57, p. 1].

2.2 Testing Techniques

Even with an classification scheme at hand (cf. Sect. 2.1.3, Fig. 2.2) it is almost impossible to exhaustively describe the available testing methods. Not only is their number high but particularly less used methods are not uniquely named. Moreover, the discriminatory power between several techniques is not always given. While one author might describe a couple of techniques as distinct ones, they could be subsumed by another one and seen as mere variations.

On this account, the (quite) brief compilation of techniques in this section serves three purposes:

- Firstly, to give an overview by introducing at least one technique per proposed category.
- Secondly, to be comprehensive in that the most widely used methods are explained.
- Thirdly, to be the foundation for later chapters of this book.

[4] It has to be noted that not all computer scientists are convinced of the usefulness of such models or at least the way they are applied. Proposing a "software process immaturity model" [48, 49] is not only humorous but also hints to skepticism.

2.2.1 Static Techniques

Static techniques consider the program at a given point, usually without executing it. They can be subdivided into *verification* and *static analysis*. Static techniques give direct hints to defects rather then identifying failures [25, p. 251].

2.2.1.1 Verification

Verification techniques aim at formally checking a program against its specification. Some of them can even prove the correctness of a program. This cannot be done with other testing techniques [5, p. 56].

Formal techniques pursue this in the literal sense. Based on the specification of the program, consistency of the program and its specification is proven [2, p. 321ff.]. Consequently, an exhaustive specification is required and it has to be transferred to a formal specification. It is not possible to validate [58] a program using formal techniques—in fact, they are not suitable to detect problems in the specification. Formal verification techniques are only used for programs in fields such as power plant control, aviation, or military [25, p. 285]. A prominent example is the application for the metro transportation system of Paris [59] based on B models [60].

Proving the correctness of a program can for instance be done using the axioms of the *(Floyd-)Hoare Logic* [61, 62] and by following *inductive assertions* [63]. A concise introduction is given by Roitzsch [25, p. 287ff.]. Hoare Logic also is an important part of Formal Specification methods; therefore, it is part of the research on e-assessment included in this book. More details are given in Chap. 6. According to Wallmüller [18, p. 218] an alternative to Hoare Logic are approaches based on *abstract data types* (e.g. [64], also cf. [65]).

An extension is *symbolic model checking* which utilizes automata theory [2, p. 45]. Formal techniques are very powerful. However, the effort is tremendous and often impractical even for small programs. Thus, they are rarely used by practitioners [2, p. 45].

Symbolic tests can be seen as a bridge between dynamic techniques and formal ones. In contrast to dynamic techniques they do not rely on samples but can draw full conclusions. The source code or an intermediary code of the program under consideration is executed symbolically. This means that variables in the program are treated as logic variables instead of parameterizing them with constant values. Commonly, an environment programmed for this purpose is used. With the given statements or instructions of a program, a *symbolic execution tree* can be built [25, p. 309f.], which is similar to a control graph. In the next step, paths trough this tree are determined (*path-sensitivity* [4, p. 404]), and the parameters required for program execution to take this path are described. Combinations of input parameters and expected output can be used as test cases.

For some programs, symbolic testing can prove correctness [2, p. 44]. In general, it is capable of detecting a very high number of defects [25, p. 315f.].

2.2.1.2 Static Analysis

Static analysis does not execute the program under consideration and does not necessarily require tool support [2, p. 43f.]. The code is analyzed and interpreted [66, p. 6]. However, tool usage is strongly suggested as static analysis techniques are suitable for automatization. In general, being static is both the blessing and the curse of the techniques described below [3, p. 115ff.]. Initiating static analyses is possible at an early point of development. However, in general no defects are found but only hints to them.

Style analysis (also informally known as *checking programming guidelines* [3, p. 122f.]) is a technique that checks source code for its compliance with predefined conventions. These conventions can be corporation-wide guidelines as well as best practices for programming languages [2, p. 271ff.]. Due to the availability of plug-ins, style analysis can seamlessly be integrated into an integrated development environment (IDE) [67]. Style analysis is particularly recommended for embedded systems but also advisable in general [2, p. 275]; moreover, it can be a good choice to maintain so called *legacy* systems [18, p. 217]. It can be expected that programmers get used to the mandatory programming style after a while. Besides establishing a corporate programming guideline that harmonizes programming styles [67], styles provided by programming language vendors (for instance [68]) and in practitioner textbooks (for instance [69]) should be followed. A comprehensive introduction is given by Kellerwessel [70]. It has been noted that style analysis should be combined with using a modern programming language, which—if correctly used—reduces the risk of programming in a *bad* style [18, p. 217].

Despite the fact that *slicing* can be done both statically [71, 72] and dynamically [73], it is attributed to the static analysis techniques [2, p. 285]. Slicing is used to automatically discover interrelations within programs, for examples with regard to the influence of statements on variables. This helps to conclude which part or even which statement of a program causes an undesired action. Patterns of defects can be identified [14, p. 178]. Whereas static slicing does not require the execution of a program, dynamic slicing takes into account information that become available during execution only [2, p. 289ff.]. Slicing can be used to better understand a program and to support finding defects [2, p. 285ff.]. Various slicing strategies exist [14, p. 179ff.]; an overview of past and current research threads is given in [74].

Control flow analysis statically checks code in order to detect anomalies. The goal is to detect code that cannot be executed for semantic reasons (*dead code*) and loops that cannot be left once entered [3, p. 118ff.]. *Call graphs* take this concept to a higher level. They document how modules are linked to each other and help to plan tests [3, p. 126ff.].

Data flow (anomaly) analysis—or *dataflow anomaly detection*—is used to detect parts of programs that deviate from the expectation. Similarly to the work of a compiler it tries to detect peculiar parts of programs; in theory, these parts could also be identified by carefully working humans [2, p. 292ff.]. The typical attempt is to check variables for being defined, referenced (i.e. read), and dereferenced (i.e. set to be undefined). After checking the program, anomalies such as redefining variables

without reading them, defining and immediately dereferencing them, or dereferencing and then trying to read a variable can be detected [2, p. 293ff.]. Compiler support for this kind of analysis is strongly recommended; explicit invocation of data flow analysis is reported to be rather uncommon [2, p. 302f.].

Control flow and data flow analysis can be combined with tools that automatically scan the code for anomalies. They can be based on various assessment techniques and might also offer visual support [14, Chap. 11].

Software inspections and *code reviews* are an extensive field with a number of techniques. It is known since the early days of testing [75]. The general idea is to manually assess the source code. Not only defects are searched for, but reviewers assess whether the desired level of quality has been reached and whether conventions are fulfilled [18, p. 201]. Well conducted reviews are very cost-effective and significantly contribute to the overall assurance of software quality [3, p. 303ff.]. Liggesmeyer distinguishes between three main techniques [2, Chap. 9]: *commenting reviews*, *informal reviewing sessions*, and *formal inspections*. Commenting reviews are the simplest technique; programs, modules or documentation is assigned to reviewers who check and comment it. Informal reviewing sessions are commonly organized as a structured walkthrough. Developers show their work to reviewers who directly comment on them. Formal inspections are the most sophisticated technique. They are not mere walkthroughs but have formally defined phases and roles. They are highly structured, which increases the effort required to conduct them but greatly increases the outcome.

A review's success is based on the participants and their interplay [18, p. 216f.]. Typically, the number of participants per review is between four and ten; participants take different roles and usually one of them is the *moderator* [25, p. 251ff.]. It can be argued that reviews are beneficial for the working atmosphere and contribute to the employees' education [4, p. 384]. The corporate culture has to be adequate for reviews, though [76, p. 119].

More detailed subclasses for reviews are e.g. suggested by Bath and McKay who (among others) mention *technical reviews*, *management reviews*, *audits*, and *design reviews* [3, p. 310]. Audits are typically less technically driven than reviews [18, p. 198ff.]. It has been noted that reviews can be more effective in removing defects than (dynamic) testing [77, p. 108] and lead to immediate improvements of the code [18, p. 201]. According to Liggesmeyer, code reviews are widely used by practitioners [2, p. 305].

A technique that is similar to reviews and that can be combined with reviews is the identification of so called *anti patterns* [78]. Whereas *design patterns* describe proven strategies to implement recurring functionality of software [79], anti patterns describe how *not* to implement functionality [80]. Anti patterns are also referred to as *code smells*, i.e. parts of source code that have a "bad smell" due to potential problems [81]. Corresponding constructs can be found both manually and with tool support. The code should then be *refactored*, i.e. changed to a better style while preserving its functionality [82]. It is recommended to use refactoring to change the code to adhere to patterns [83]. Some code smells are also found by the above described techniques. For instance, dead code can be identified by control flow analysis.

Metrics describe key figures that are generated from source code, intermediary code, programs, program documentation, or systems that manage testing related data. Metrics aggregate data and allow to draw conclusions, or to estimate duration or time requirements. They are used to assess a program's quality, to handle complexity, to control the development process, and to check if standards are met [2, p. 232f.]. Furthermore, they are useful to estimate effort, costs, and duration, to proactively identify problems, for reasons of comparison, and to judge the effectivity of newly introduced tools and methods [2, p. 232f.]. They can also help to find *problematic* areas in the code [3, p. 362ff.]. Code that becomes too complex due to nested statements, methods that are very long, and similar criteria hint to potential problems.

Some metrics are also important for the operation of testing tools. For example, the percentage of covered control flow graph edges and def-use chains can be calculated [84]. However, metrics in general only take a subordinate (or rather auxiliary) role in testing. Therefore, no introduction into different kinds of metrics and their usage is given. Please refer to either [2, Chap. 7] or [24, Chap. 11].

In addition to the above mentioned techniques, manual inspection of diagrams can be seen as a static technique [2, p. 276ff.]. This e.g. applies to control flow graphs, Nassi–Shneidermann diagrams [85], and structure diagrams.

2.2.2 Dynamic Techniques

Dynamic techniques are based on executing the code in order to detect failures. They comprise a high number of techniques which can be put into six categories. Descriptions are given in the following sections.

2.2.2.1 Structure Oriented

Structure orientation means that a program is tested against its own code [66, p. 8ff.]. It forces testers to check the code in detail. The effort and the required skill of employees for structure oriented techniques are high but it is possible to effectively find defects [3, p. 77ff.]. The main criticism is that the structure itself might be incorrect, for example as a result of incorrect requirements. While other techniques might at least help to reveal this, it cannot be detected based on the structure [3, p. 81]. Advice on choosing functional oriented techniques is provided by Bath and McKay [3, p. 94ff.].

There is a number of structure oriented techniques that are very simple. Since they are inferior in results to control flow testing, they will not be discussed in detail. For *statement tests*, as many test cases are created as are needed to cover each (source code) statement at least once. *Branch tests* require each branch—i.e. each transition from a statement that involves a decision—to be covered at least once. Various kinds of *condition tests* require atomic conditions or combinations of them to be covered [3, Chap. 5]. Condition tests that are derived from source code can

also take the evaluation of statements by compilers into consideration to reduce the number of test cases [5, p. 161ff.]. For example, in most object oriented programming languages *bitwise or* (the | operator) is evaluated completely, whereas *logic or* (the || operator) does not require complete evaluation if the first operand is `true`.

Control flow testing tries to cover each path through a program. For this purpose, a control flow graph (CFG) is built [2, p. 85]. It is set up by checking each program statement in the source code or instruction in the machine or bytecode. Each statement/instruction forms a node in the graph, whereas transitions make up edges. Simple statement such as $x = 2$ just have a transition to the next statement; conditions have two (e.g. `if (...)`) or more (e.g. `switch(...)`) transitions. A technique to create CFGs for Java is described in [84]. With the given CFG, test cases are created that cover each transition at least once. The set of test cases is tried to be kept as small as possible. Alternatively, a threshold for the desired coverage rate can be defined [3, p. 92]. A reasonable threshold can often be reached with a much smaller number of test cases than were needed for full coverage. This is caused by test cases that cover multiple paths trough the program whereas some transitions require tailored test cases to cover them. Selecting a threshold might even be a necessity. Due to semantic restrictions not all edges might be coverable [4, p. 259f.].

It is possible to subdivide control flow testing, e.g. into *path testing* and *loop testing*. The latter is also known as linear code sequence and jump (LCSAJ) [86] or jump-to-jump path (JJ-path) [87] [88, Chap. 10]. This is not done in the recent literature, though. Some lesser known techniques are explained in [25, p. 408ff.]; different coverage criteria are discussed in [7, p. 73]. A comparison of control flow based techniques is presented in [4, p. 258f.]. Sometimes, path testing also is used as a synonym for control flow testing (e.g. in [3, p. 92]).

Data flow testing uses information on how data is processed by a program. For this purpose, it is checked when variables are read and written [2, p. 142]. This can be done on a source code level but also for intermediary code or machine code. For intermediary or machine code that is run on a stack machine, it becomes more complex because loading a variable onto the stack does not imply that it is read.

A very effective way to test the data flow is to check the coverage of full definition-usage chains (*def-use (DU) chains*, sometimes referred to as *set-use pairs* [77, p. 293ff.] or *all DU-paths* [2, p. 148f.]). They can be defined as follows [84]:

X: variable, S, S', S'': statement or instructions
$def(S)$:=$\{X|S$ writes to $X\}$
$use(S)$:= $\{X|S$ reads from $X\}$

If $X \in def(S) \bigcup use(S')$ and $X \notin def(S'')$ for each S'' on a path from S to S', (X, S, S') forms a def-use chain. In other words: there is a def-use chain between two statements/instructions if the first one defines (i.e. writes) a variable and the second one reads (i.e. uses) it without another statements/instructions changing the variable in the meantime. Similarly to control flow tests, data flow tests try to find test cases that cover all or a threshold of all DU chains. For semantic reasons, covering all chains is impossible for some programs. Data flow testing can reveal defects that are not

detectable with control flow testing. However, it can become much more complex. In particular, intraprocedural DU chains can be created (e.g. to cope with *aliases* [4, p. 109]). Whereas intraprocedural CFGs are built by simply connecting graphs at the statements/instructions that invoke and leave methods, chains that reach among methods have to be generated explicitly [84]. For the intricacy of interprocedural data flow testing also cf. [4, p. 109ff.]; special attention also has to be given to arrays and pointers [4, p. 106ff.].

Besides full DU chains [4, Chap. 6], many weaker criteria for data flow testing are known. Examples are *all defs* and *all uses* (if desired limited to some kinds of usage) [2, Chap. 4] [25, p. 433]. In general, they base upon weaker assumptions and are more convenient to find and to cover. However, the stricter the criterion is the more likely are defects to be found. How criteria subsume each other is given in a graphical overview by Roitzsch [25, p. 447f.]. Liggesmeyer also describes two additional techniques, the *required k-tuples test* [2, p. 163ff.] and *data context coverage* [2, p. 171]. An approach of testing based on coverage has been suggested for testing with only a very limited amount of available time [16, p. 227f.].

2.2.2.2 Function Oriented

To test a program's functions, cause and effects of it are taken into considerations. While structure oriented techniques focus on the program itself, function oriented techniques have a higher level view [66, p. 234]. Testers read the specification documents and design corresponding test cases [2, p. 50]. Therefore, function orientation can also be called *specification orientation* [3, p. 31]. Test cases are derived in a systematic way rather than choosing a *brute force* approach [4, p. 189].

Testing with function oriented techniques does not necessarily lead to full testing coverage but helps to ensure that specifications are met [3, Chap. 4]. Despite not being the primary aim, function oriented techniques can also reveal inconsistencies or even mistakes in the specification [5, p. 47f.]. In general, function oriented tests of modules should not be conducted by the developers that implemented them [7, p. 83]. Advice on choosing function oriented techniques and a discussion of their practical relevance is provided by Bath and McKay [3, p. 60ff.].

The most simple function oriented technique besides random testing (see Sect. 2.2.2.6) is testing for *functional coverage*. It is tried to cover each function with one test case [25, p. 334ff.]. Function coverage is not only inefficient but also faces severe limitations when testing object orientation programs [25, p. 337].

So called *use cases* depict scenarios in which a program is actually used [3, p. 59]. They are used in requirements analysis and specification. Some specifications already contain prepared use cases [5, p. 141]. The Unified Modeling Language (UML) provides *use case diagrams* for visualization [89]. Each use case is made up of a number of paths through the program. For each use case, as many test cases have to be created as are required to cover each corresponding path at least once [3, p. 59f.]. It can also be organized as *thread testing* where testers assess functionality in an order that end-users would typically chose [90, p. 22]. Use case testing is

customer oriented, but it is very likely that tests are left out [3, p. 59f.]. Functionality will be untested if the usage of a program differs from the developers' intention.

Decision tables are special forms of combinatorial tests. They are useful to describe business rules [77, p. 174]. Decision tables summarize rules in clearly arranged tables. Each rule is made up of *conditions* and *actions*. Each combination of conditions triggers a number of actions. Typically, conditions take the values "applies", "applies not", or "irrelevant" [26, p. 94ff.]. Actions might occur (or not). One test per table column is needed [77, p. 176]. However, full coverage of the rules is possible. Consequently, it is easy to test with a given decision table and the technique is very popular; however, creating the table is challenging or even almost impossible if only imprecise requirements for a program are available [3, p. 44f.]. Besides, tables grow exponentially in the number of conditions [2, p. 79]. Decision tables are used for *cause effect analyses* that try to decrease the number of required test cases [5, p. 125ff.]. It also helps to graphically depict a cause and effects graph [25, p. 352f.], which is built based on a program's specification [18, p. 232].

Equivalence partitioning (in older literature also called *input testing* [88, p. 57]) is a very common approach. It is almost universally usable [2, p. 289ff.]. Instead of testing randomly, test cases align with classes that are to be tested. This is done both in a macro and in a micro perspective [77, p. 144]. For example, parts of a program are chosen for being tested based on their characteristic. In a module of the program, equivalence classes are created that directly address its elements. There are two kinds of equivalence classes: *valid* and *invalid* ones. Think of a text field that takes values for full hours of a 24 h clock. -1 and 31 are examples for invalid classes. $0-23$ is the class of valid values. For testing, *one* representative value from each class is taken [5, p. 123f.]. Testing could be done by having three test cases for -1, 5, and 28. Of course, other values are possible. Equivalence partitioning is an efficient technique but prone to leave out tests [3, p. 35ff.]. This usually is caused by accidentally creating classes that subsume more than one class. It also has to be noted that equivalence classes in general are not disjunct [4, p. 188].

Using numeric values (in particular intervals) is sometimes described to be *boundary values analysis* whereas equivalence partitioning only uses textual or property values [77, p. 144ff.]. Boundary values have been found to be related to failures [7, p. 90]. To set up boundary values, classes have to be partitioned beforehand [3, p. 38]. Equivalence partitioning with alphanumerical values can also be called *syntax testing* [88, p. 85].

It is recommended to test *special values* along with a boundary value analysis [25, p. 346]. Values such as zero, the maximum value for a variable, empty data structures, `null` references, so called *magic numbers*, and reserved statements might trigger special routines or lead to other problems. A list of typical special values is for instance given in [25, p. 347].

Due to the high number of combinations in large tables, approaches have been found that increase efficiency. *Domain testing* is used to check equivalence classes' boundaries and reduce the number of test cases required to test all (remaining) classes. They not only use functional equivalence classes that were derived from the specification but also structural equivalence classes [25, p. 392]. Several examples for

domain testing are given by Black [77, Chap. 16] and Roitzsch [25]; examples are *path domain tests* and *tests of code that hints to defects*. Additional techniques for the same purpose are *orthogonal arrays* (cf. [91]) and *all-pair tables* [77, Chap. 18]. Both mean to reduce the numbers of test cases required to satisfyingly test with equivalence classes. In general, this can be done with success but there is the risk to omit required test cases [3, p. 48f.]. Graphical support for this task is provided by drawing *classification trees*. They visualize the pairing which can increase lucidity—or alternatively lead to immense trees with the opposing effect [3, p. 58].

Another approach towards combinatorial testing is to take equivalence partitioning as the basic idea and to derive concrete techniques from it. *Category partitioning* as described by Pezzè and Young [4, p. 204ff.] basically resembles the above description of equivalence partitioning. They also propose techniques for optimization such as *pair-wise testing* (also known as *all pairs* [92]) [4, p. 215ff.] and *catalog based testing* [4, p. 219ff.]. Therefore, it has to be noted with regard to the literature that combinatorial techniques are introduced in a variety of distinctions and with a variety of names.

Partition Analysis is an approach that combines different ideas, namely verification and testing [93]. It can be attributed to various categories [2, p. 206]. Partition analysis uses symbolic execution and defines three criteria for comparing specification and implementation of a program: compatibility, equivalence, and isomorphism [2, p. 206f.]. If verification of these partly fails, additional tests are conducted [2, p. 207f.]. More information is given in [94]; a newer approach is described in [95].

A technique that takes the idea of structured testing to black-box tests is *state oriented testing*. It sometimes is attributed to the model based techniques [4, p. 129]. Without knowing a program's code its possible states are figured out. The aim of testing is to cover each transition between states at least once. This is easy and efficient for programs that have known states and a small number of transitions [3, p. 48]. In contrast to other functional tests, a program's memory (i.e. its state) is taken into account [2, p. 58]. If a program can be described as a finite state machine, tool support is available [5, p. 135]. Nevertheless, manual effort cannot be completely circumvented [4, p. 130].

State oriented testing is not applicable if the number of states is unknown or if a high number (say, at least a couple of 100) of transitions has to be covered. The *state space explosion* quickly becomes problematic for human testers, but even machines cannot keep control of the states that generally grows exponentially with the size of a program [2, p. 64]. Therefore, it is mainly used to test software for embedded systems [3, p. 49]. Based on tests with state machines, *statistic state oriented tests* have been developed. They take into account the probability that a function is used and try to find those defects first that are more likely to be encountered during an actual usage of a program [25, p. 363ff.].

It is not only possible to test function *oriented*, but to test the functions of a program. This is called *functional testing*; however, the term is potentially misleading. The technique aims to check whether the software adheres to its specification [3, p. 141]; this task is conducted completely manual. Besides, it can be argued that a

program is rather *checked* in order to determine if it complies with the requirements instead of being *tested*. Typical techniques are *checking for correctness*, *checking of commensurability*, and *interoperability testing*; a detailed introduction is given in [3, Chap. 10].

2.2.2.3 Dynamic Analysis

Dynamic analysis techniques can reveal defects that are almost impossible to find with other techniques [3, p. 128]. The kind of failures encountered can be critical and at the same time hard to reproduce. Resource problems are likely to occur after a software has been finished. Such defects are very expensive to fix [3, p. 128f.]. Besides finding defects, dynamic analysis can be used to visualize a program's performance and to deduce runtime information. Dynamic analysis might slow down execution, and it is dependent on the programming language used [3, p. 129f.].

Memory leaks describe a conditions in which programs continuously allocate more memory than they need. In most cases, memory is correctly allocated but not completely deallocated after not being needed any more [96, p. 254]. Subject to the program, memory leaks can be annoying to fatal. In some cases they are not noticed, in others they slow down execution or even lead to a crash of the executed program. They are hard to reproduce since it can take a long time (in extreme cases months) before they have consequences, and they are usually hard to fix. They are more common in programs that have been developed with programming languages that do not use garbage collection[5] [4, p. 406ff.]. Tool support is highly recommended for detecting memory leaks [3, p. 131].

Software that consists of multiple large modules or that has sophisticated algorithms might be subject to *performance bottlenecks*. They either occur if a program heavily uses distinct parts of the hardware (for example the mass storage system) or if single components of it have a by far inferior performance compared to others. In both cases, analysis tools can help. They supply information collected during execution and can also depict them graphically [3, p. 135]. A *resource monitor* could for example show that the mass storage system is often accessed randomly or that a program uses many CPU cycles at certain times of execution. Both might be countered with adequate strategies. Also, slow algorithms (e.g. for data structure access) can be detected. Other tools can reveal communication deficits. An example would be synchronous messages that block execution due to high latency. Once detected, it can be checked whether replacing them with asynchronous messages is possible.

Dynamic symbolic execution sometimes is attributed to symbolic tests (see Sect. 2.2.1.1). The basic principles are the same: variables are treated as logic variables (also referred to as *logic expressions* [18, p. 219]) instead of parameterizing them with constant values. However, in place of statically checking the code, it is executed—hence dynamic symbolic execution. Since information that is only

[5] Garbage collection automatically frees memory and deallocates data structured not used anymore [97, 98].

available dynamically can be used, results are more precise. For example, *dynamic binding* and *virtual invocation* can be taken into account. This can be extremely complex, though [4, p. 405]. Besides, techniques known from logic programming can be utilized since the symbolic execution tree is used as a *search tree*. An example is *backtracking* [99, p. 90]. The test case generation (TCG) tool presented in this book, Muggl (see Chap. 4), is based on dynamic symbolic execution.

There are further dynamic analysis techniques that try to discover programming problems that cannot be detected statically. An example for a rather often described technique is the detection of *wild pointers* [3, p. 132ff.]. They are also called *dangling pointers* (cf. [100]). Pointers hint to memory addresses that are used to read or store data, or as jump addresses. Incorrect pointer usage and pointer arithmetic can cause various problems. Finding (and removing) wild pointers can avoid some of them. Another example is *lockset analysis* which tries to discover *race conditions* [4, p. 409]. Race conditions can occur when multi-threaded programs use shared variables; they are hard to detect and tend to be nondeterministic.

2.2.2.4 Performance Tests

Performance tests can also be called *efficiency tests* [3, p. 171]. They are used to assess the runtime behavior with regard to resource utilization and response times. Adequate performance is not only a factor for usability but important for the overall quality of a program. Performance tests are also conducted to ensure that a program will work under unforeseeable future conditions [3, p. 171f.]. There are two main testing techniques and a number of additional techniques that rather measure than test.

Load tests are used to assess a program's behavior under load. The load, i.e. the sum of inputs to the program, resembles the typical workload that is estimated for the program. Aims of load tests include

- to check how programs handle parallel requests and how many requests they can process in parallel,
- to find out how the session management of a program works, i.e. how seamless parallel usage is handled,
- to assess the response times of the system under varying conditions, and
- to monitor resource usage in relation to the program's load [3, p. 172ff.].

Load tests can also be called *volume tests* [7, p. 124f.]. In general, load tests should be distinguished from *benchmarks* which measure a program's performance and try to generate key figures [7, p. 131ff.].

Stress tests have a different aim. Rather than using typical workloads, programs are tested with loads that are beyond the expected workload. In fact, loads are even beyond the maximum noted in the specification in order to fathom how the program behaves under such workloads. In general, programs should behave *graceful*. Instead of an abrupt crash, a program should be able to save opened files, persist sessions, and ensure that no data is corrupted. Ideally, a program should switch to a mode

of minimal resource usage. For example, if it is flooded with requests, a program could try to accept them as long as possible and answer them once the number of incoming requests decreases. In a Web environment this strategy might however be fatal; requests not immediately answered will—due to short *timeouts*—not be received anyway and clients will send more requests. Therefore, in this case unanswerable requests should be discarded until the load decreases. In contrast to this, a crash is undesired since it would take a while to reboot the program. If a crash is unavoidable, it should be a *graceful degradation* [3, p. 174], i.e. an emergency shutdown and not a halt at a random point.

Furthermore, stress tests can be used to locate bottlenecks in programs [3, p. 174]. This especially applies to programs that are part of larger systems. For example, a system that provides data from files on request is likely to hardly utilize CPU and memory but to create heavy I/O load on the mass storage systems. If a number of requests arrives that is hard to handle, it should not try to deliver all files at once which would degrade the mass storage's performance and eventually lead to a throughput near to zero. Instead, it should use the bountiful amount of memory and store requests. They can be processed once I/O load is lower. In addition to this, stress tests might hint to implement a memory cache for frequently requested files.

Stress tests can adhere to various strategies. *Bounce tests* vary between nominal and extraordinary loads to check whether a program adapts its resource usage [3, p. 175]. It has been argued that stress tests can also be tests that do not induce a high load but just violate the quantitative specification of a program. An example is to test a program that should accept 16 parallel users with 17 users [7, p. 128f.]. However, this rather is a functional test than a stress test.

A number of further techniques mean to assess a program's performance rather than to test it. *Scalability tests* are called tests but aim at finding out whether systems scale. Scaling means the adaption to changed patterns of usage [3, p. 176f.]. A scalable program could simply adapt to an increased workload, could perform more requests on upgraded hardware, or be prepared to spawn multiple instances on different machines. This is checked with regard to the estimated future requirement. Checks of the *resource utilization* and *efficiency measurement* are used to monitor a program's performance and to derive key figures from its execution. Key figures can be response times as well as throughput figures and similar data [3, p. 178ff.]. They help to learn how a program works and which parts of it might need improvement. One challenge that particularly concerns monitoring of programs is to avoid a bias induced by actually monitoring them [3, p. 180]. Measurement results can considerably be altered by the very act of measuring.

All performance tests should be done with tool support [3, p. 188ff.] [101, p. 195f.]. Measurements require tools or at least specialized routines within programs in order to receive the data needed to calculate the key figures or to monitor resources. Load and stress tests could theoretically be done with a (very) high number of testers but having specialized employees simply generate load by e.g. clicking buttons or continuously reloading Web pages is extremely inefficient (and, by the way, frustrating).

Besides testing its performance, a program can be checked for its reliability. Such *reliability tests* incorporate elements from load tests and manual assessment. They are used to fathom whether a program will fulfill its intended tasks without the risk of outages or undesired behavior [3, p. 233f.]. While performance tests address the actual performance, reliability tests e.g. check whether *failover* routines work correctly, backups are created as planned, and restoring data is possible [3, Chap. 14]. The latter can also be called a *recovery test* [7, p. 129f.]. Reliability can be assessed both with regard to functionality and time figures. Concerning failover functionality, it can be checked whether a program is tolerant towards failures. Ideally, failures should not disrupt the core functionality but be handled by the program as good as possible. Methods commonly used are *failure mode and effects analysis* (FMEA) [102, 103] and *failure mode and effects and criticality analysis* (FMECA) [104].

2.2.2.5 Experience Based

Experience based techniques do not rely on a structured approach but utilize a tester's experience. Thus, they are also called *free* testing techniques [5, p. 169f.]. Testers need to have a lot of experience and high testing skills in order to test effectively [25, p. 366]. The reproducibility of the tests is low.

Intuitive testing is an almost random approach where testers test whichever functions they think should be tested [3, p. 106]. It is also called *error guessing* [5, p. 170ff.]. Testers have in mind where defects might be or what might cause failures (*error hypotheses*). Thaller calls this their "sixth sense" [7, p. 93]. With the created test cases they check whether the guessed errors actually exist. To steer this technique, *checklists* can be used [3, p. 108]. Additionally, testers can rely on so called *bug taxonomies* which classify defects [5, p. 171f.]. Knowing typical defects helps to derive error hypotheses.

Explorative techniques are characterized by the fact that testers do not know the program they test but rather explore it. Finally, *attacks* are used to check whether a program is prone to common problems. It often is used for security testing (cf. with the following section). If qualified testers are available and testing has to be done in a short time, experience based techniques can be very effective. With a greater number of testers it can become inefficient, though, and some defects are likely to stay hidden. [3, p. 108f.]

2.2.2.6 Further Techniques

Dynamic testing is heuristic and accepts that test case generation is fuzzy to some degree [25, p. 375]. *Diversifying* techniques try to overcome this fuzziness. The two main methods are back-to-back tests and mutation tests.

For *back-to-back tests*, two or more programs are independently built by multiple development teams based on the same specification. They are intended to be semantically equal [25, p. 376] (i.e. redundant [2, p. 180]). For testing, the programs are

run and their outputs are compared. In general, divergences are reported to all development teams in order to check for their cause. Testing is iterated until all programs behave equally. Back-to-back testing can dramatically reduce the number of defects but is costly [25, p. 383]. Nevertheless, it can be cost-effective [105]. Back-to-back testing is especially used for embedded systems [2, p. 184].

Mutation tests do not mean to test a program but the tests written for this program [7, p. 130]. They are attributed to *fault based* testing [4, p. 353ff.]. So called *mutants* are derived from the original program. Mutants are marginally changed and then tested with the original test cases [106]. It can be said that errors are "injected" [101, p. 197f.]. The quality of test cases is reflected by their ability to identify mutants [25, p. 384]. Good test cases are likely to reveal new defects [7, p. 85]—and the mutants. The changes in mutants follow typical programming mistakes such as typos, wrong usage of brackets, and inverted operators [25, p. 384]. *Strong mutation* tests cover the whole program while *weak mutation* tests [107] cover only some components. To increase testing efficiency, mutation tests can be parametrized. However, mutation tests are only applicable with tool support [25, p. 390]. It is estimated that a program of n lines of code requires n^2 strong mutation tests [108].

It is possible to test completely *random*. The difference to error guessing is that there is no intuition that a tester follows, but test cases contain random parameters. This usually is done by using so called test data generators [25, p. 194f.]. The surprising effectiveness of this technique has been demonstrated with Quickcheck, a tool originally developed in Haskell [109–111]. Ports or tools inspired by it exist for the most common programming languages such as Java [112, 113] and C++ [114].

Statistic Testing is based on probability distributions of the usage of classes, methods, or other program components. Tests are generated and their contribution to covering classes, methods, or other program components is assessed. It is specifically used for system tests in which the reliability of a program is checked [25, p. 368]. If statistic testing is done with uniformly distributed tests, test generation equals random testing. This also can be called *stochastic testing* [2, p. 517].

Instead of specifying test cases, programs can be tested with *original data*. This can only be done if very similar programs exist or the tested program is an upgraded version of an older program. The idea is to record data formerly fed to a program and to use it as input for the program to test [26, p. 97ff.]. This kind of testing seems effective because it uses realistic input data. The code coverage of such tests is reported to be low, though [25, p. 195]. This might be improved by carefully selecting the data. Additionally, old datasets might be used as input data for load tests and stress tests [26, p. 97ff.].

2.2.3 Testing of Web based Programs

In recent years, *testing of Web based programs* has become important. Not only has the number of such applications increased but the numbers of them in comparison to classical *rich client* applications has also become much greater. With the emergence

of the so called Web 2.0, Web based programs have gained complexity. They are expected to be as comfortable to use as rich client programs. The interaction of client-side scripts such as JavaScript and server-run application parts add further complexity.

In general, most of the techniques that have been described also apply to software that is used on the Internet or accessed via Internet based protocols. Some particularities have to be taken into consideration that arise from the architecture. Often, the model-view-controller (MVC) design pattern [115] is used. Consequently, programs consist of a number of layers that take distinct tasks and encapsulate functionality. Testing the *presentation layer* might for examples reveal failures that are caused by the *business logic layer*, but it will be very hard to locate them. These particularities apply to other multi-layered applications, too. The same applies to load and stress tests. There are some particularities, but they are not only found in Web based software.

Fields of testing that only apply to Web based programs address the interaction of client and server side. For example, programs can be tested with regard to their output (e.g. Web browser compatibility [6, p. 820]) or the interaction of client side and server side code. In particular, partly loading Web pages using asynchronous JavaScript and XML (Ajax) is a challenge. Some tests require specialized tools to perform them with Web based programs (cf. Sect. 2.4).

Especially Web based programs that can be accessed via the Internet can be target of attempts to compromise them. Hence, testing of Web based programs includes security tests (see Sect. 2.2.2.6). Many security tests can be parameterized for Web applications.

2.2.4 Additional Techniques

Not all techniques fit into the chosen categories. Above all, some techniques cannot be categorized because they apply to all or most categories, or amend techniques with context related additions. Such techniques are discussed in this section. Additionally, some techniques are mentioned that might be categorized as non-functional testing techniques; however, in contrast to e.g. performance tests they are rather *checks* than tests by their character.[6]

Object orientation and testing has gained increasing attention in the last ten years. Most testing techniques either originate from the era of imperative programming or are independent of a programming paradigm. Thus, trying to optimize them for testing object oriented programs and to reconsider their applicability is advisable [2, p. 408]. Apparently, object orientation has not simplified testing [116] but rather risen the level of complexity [66, 117, 118]. The most notable differences of

[6] A comparison of tests and checks is given in [5, Chap. 7]. Unfortunately, classifying techniques as tests and checks is ambiguous.

object oriented testing concern the phases of module and system testing [4, p. 306ff.] [66, p. 21ff.]:

- Due to modularization, there are more interprocedural interdependencies. This does not only rise complexity but also means that defects might arise due to the interplay of modules developed by several developers.
- Sharing of class members (*fields*) might couple methods of classes that are not meant to or at least do not appear to share data. Method results might be subject to a class' state.
- Programming for reusability and a more generalized functionality of classes introduces more possibilities to test them. In particular, testing has to go beyond the currently required functionality of a program. As a consequence, paradigms such as *test first* demand only explicitly specified functionality to be implemented.
- There is no linear growth between the size of the class and its complexity. In fact, its complexity rises faster.
- Code instrumentation (i.e. the usage of code for a given analytic purpose) is more complex. Dynamic binding complicates static testing methods [7, p. 166].
- Modularization, inheritance, and polymorphism have to be given special care. Manual inspection becomes burdensome [2, p. 409].

Techniques to overcome the above mentioned challenges are diverse and numerous. Detailed remarks on using testing techniques for object oriented programs are given by Sneed and Winter [66], and by Vigenschow [101, Chap. 9]. Liggesmeyer [2, Chap. 13] gives advice how to modify existing techniques; Pezzè and Young present novel approaches [4, Chap. 15].

Particularities apply when programs are tested that do not follow the imperative paradigm. Examples are programs that have been built using *functional* (e.g. [119]) and *logic programming* languages (e.g. [120]). Despite the effort to write such programs, testing might be easier due to the mathematical foundation of the paradigms (cf. [76, p. 135ff.]). While the coverage in textbooks is low, testing of functional and logic programs is an active field of research (cf. e.g. [121, 122]. Details, however, are out of scope.

Embedded software requires special attention for testing. Embedded systems are not only prevalent but also used in many critical contexts such as safety technology and healthcare [2, p. 439]. In general, many of the above described methods can be applied. In particular, static methods are feasible since most embedded systems have a limited number of states and do not consist of a lot of code. It might be possible to simulate them as state machines. Extensive embedded systems often are split into separate subsystems. Ideally, testing should prove correctness [2, p. 440]. Many embedded systems are also *real-time systems*. As a consequence, parts of the software need to comply with real-time requirements in particular with regard to their response characteristics. Embedded systems are used for a variety of purposes [101, p. 222f.]. Their criticality follows their real-time behavior [101, p. 223]: in *hard real-time* systems, time violation has fatal consequences; in *soft real-time* systems, time violations are annoying but not fatal; *stable real-time* systems act *soft* until a deadline is reached. Testing thus needs to take time criticality into account. Embedded systems

have to be failure-tolerant [101, p. 224]. Even if failures occur, the system should continue its work. Testing thus has to include this functionality—embedded software has to be checked for reliability and availability [2, p. 441f.]. Besides, in many cases quantitative figures on a system's reliability are expected [2, p. 439] (for instance by the contractual partner). Due to their specialty, distinct techniques will not be discussed in this book. Introductions are given in [101, p. 226ff.] and [2, Chap. 14].

Security tests subsume techniques from various other techniques and some special methods. Typically, so called *penetration tests* are *attacks* that try to compromise a program [3, Chap. 13]. Attacks could aim at *crashing* the system, at gaining unauthorized rights for using it, on altering or deleting data, on retrieving classified data, and generally at using a system in a way it is not intended to or the user is not allowed to. Typically used testing techniques are stress tests, static analysis, and function oriented tests. The latter can be used to find program statements prone to so called *buffer overflows* [123] which can be used to insert malicious code into a program. Security testing is a field of its own and not further discussed here. An introduction is e.g. given in [124, 125]. First and foremost, security of web applications and test methods for them have gained increased attention in the last years (cf. [126, 127]). Security testing should follow checklists and guidelines (e.g. [6, Chap. 24] or [128]).

Regression testing is not a technique of its own but has to be mentioned along the other techniques. It describes the repetition of tests after the tested program has been changed [2, p. 192f.]. Regression testing should show that defects of past versions have been fixed and that no new defects have entered the program as side effects [25, p. 242ff.]. It is of no importance whether a program was changed as part of development, to add functionality, or to fix defects. Regression testing should either aim for efficiency or automatization. Automatization is a mean to increase efficiency. Regression testing relies on an adequate documentation [4, p. 483f.]. Ideally, only those tests are repeated that would find failures in changed components [66, p. 14f.]. Alternatively, all tests are repeated if this is possible without human intervention. Advice for choosing which tests are to be repeated is e.g. given in [4, p. 484ff.].

Companies often buy software instead of developing it on their own. Large companies nevertheless employ developers and testers; software these companies buy uses to be customized or at least adapted to the existing system landscape. Therefore, *testing of standard software* is seen as a technique of its own. It, however, follows many of the techniques described earlier. In most cases, no glass-box tests can be done because the source code is not (fully) available. Failures need to be documented with much care in order to report them to the company that develops the program. Testing should not only find defects but make sure that a program meets the expected reliability. At the same time, the effort should be kept low [6, p. 841]. An introduction and checklists are provided by Perry [6, Chap. 22].

Testing *usability* and *accessibility* has some characteristics of testing but could also be categorized as a *check*. It ascertains how usable a program is, how effective it works, how satisfied its users are, and how easy users adapt to it [3, Chap. 11]. Inspection and review techniques similar to those of static analysis can be used although testing is dynamic in this case.

Similar considerations apply to tests for *maintainability* and *portability* [3, Chaps. 15, 16]. It is without questions that it is vital for most programs to be maintainable and that some programs need to be portable. Nevertheless, judging the portability of a program with regard to its adaptability, installability, replaceability, ability to coexist with other programs, and its conformity with standards (*portability compliance*) [129] is not really a test. Testing for maintainability should be accompanied with an *impact analysis* that shows which part of a program would be affected by e.g. a migration to another platform [5, p. 103ff.]—this analysis is *not* a test. Consequently, for this work these kinds of tests are seen as checks despite their naming as *tests*. An exception are portability tests in multi-platform environments. They assess whether programs that exist in versions for multiple platforms behave in the same way [6, p. 876f.]. This is done by using techniques as described earlier and comparing the results.

Another testing-related activity that not really is a test is examining a program's documentation [6, Chap. 20]. Checking the documentation is vital; creating and maintaining it accounts for 10–25% of the total development costs of standard software [6, p. 797]. Writing a documentation cannot be compared to writing a computer program, though. Incorrect specification will be reflected in the documentation and there are typical mistakes that are done when writing it; nevertheless, it does not have the technical character of programming and programming related activities.

Finally, quality assurance of the specification and of requirements rather is a management activity than a check. Nevertheless, it often is mentioned in textbooks about testing. An introduction is for instance given by Roitzsch [25, p. 258ff.].

2.2.5 Techniques with Regard to Phases

While the above discussed techniques can be used in all phases of testing, they especially apply to the first phases. Technically speaking, tests conducted in different phases of testing are no techniques of their own. However, some techniques are rather used in one phase than in others. Besides, some particularities apply to some phases. Therefore, it for instance is spoken of *integration tests* as if it was a technique. More details are e.g. given in [4, Chap. 22].

For integration tests it has to be decided whether all components are integrated at once ("big bang integration" [77, p. 309]) or if components are added (and tested) successively. The latter can be done in a bottom up, top down, or backbone (simplest functions first) approach [88, p. 113f.]. A distinction is also possible by *incremental* vs. *not incremental* and *dependency oriented* vs. *function oriented* strategies [5, p. 23ff.]. Integration tests are done when distinct components have been sufficiently tested [5, p. 23] but should not be postponed too long [96, p. 267]. They can follow the same techniques that are applied to a broader set of by then integrated components. In general, integration tests are less technically driven than component tests [26, p. 53ff.]. Consequently, rather manual testing methods are chosen.

System tests are integration tests on a broader level. They use the same methods, but the focus is even less technical. They consist of a high number of distinct tests [25, p. 75]. Sometimes, advanced system tests that try to fathom whether a system can be considered to be *finished*, are called *smoke tests* [130, Chap. 17].

Similar to integration testing, *acceptance tests* are rather a phase than a method. Their particularity is a focus on black-box tests and the involvement of customers or other non-technical personnel that is hoped to accept the program. The aim *should* be to highlight the program's quality and to get user feedback instead of finding defects.

2.3 Organization of Testing

Testing is based on technical methods but it also has an organizational dimension. Successful testing is bound to preconditions that have to be met in companies that conduct software tests. Test organization comprises of seeing testing as a process and dividing it into phases, of documenting tests, of test management, and of certifications for testers. The first two tasks are even important for academic software development, or software development by single users.

2.3.1 Phases of Testing

If testing is meant to be effective and efficient, it is not just done at one fixed point in the development process. It neither is static but the testing techniques should adapt to the status of the program under development. By no means, software development and testing should follow a "code and fix" approach [77, p. 28f.].

Therefore, testing is done in phases. This partitioning is called *staging* by practitioners [26, p. 13]. There is no definite scheme for the phases used; but there is a typical layout which can superficially be followed or become a corporation-wide standard. The partitioning proposed roughly resembles the classical V-Model [40, 131] and has been described in [132] and subsequentially in [133].

1. *Component test* (also known as *module test* or *unit test*)
   Developers test modules they coded. They work both incremental and iterative. Testing responsibility remains with the developers even though there can be guidelines on how to test. Usually, the level of documentation is low. Since developers know their own code, they conduct Glass-Box tests and probably also Gray-Box tests. Black even distinguishes between component and unit tests: unit tests concern "an individual piece of code" whereas component tests already deal with "pieces of the system" [21, p. 5f.].
2. *Central component test*
   In contrast to the first phase, this optional test is usually not done on the developers' working stations. Components are tested on a system which is similar

to the target system for the finished software product. Developers are usually supported by testers.

3. *Integration test* (also known as *product test* [21, p. 6f.])
   This test usually is not done on developers' working stations but on a testing system. Components developed by multiple developers are integrated and their interplay is checked. In general, developers are still responsible for this test [18, p. 224] and it is finished with a formal approval. Developers assure interoperability; if integration related problems are found in a later phase, interoperability has to be fixed and the integration test repeated.
4. *Performance test*
   A first, optional performance test can be conducted whilst or shortly after the system test. It aims at getting a first idea of the system's performance and at finding possible performance bottlenecks.
5. *System test*
   The system test is a large-scale integration test which is less technically focused. No more Glass-Box but only Black-Box tests are used. *Test stubs* [101, p. 180f.] and so called *mock objects* [134], which simulate behavior of missing components in earlier phases, are successively removed. Testing commonly is done by dedicated testers who do not know the source code. Testing should be documented with much effort and the phase be finished with a formal approval that assures the program's quality to be at a level that satisfies the customer. System tests are regarded as particularly important—there even are textbooks that solely deal with them [135].
6. *Performance test*
   As soon as all main components are integrated, another optional measurement can be done. It is driven functionally and not done by developers. Whereas the first performance test could reveal algorithmic limitations and general misconceptions, the system performance measures the program under realistic conditions.
7. *Acceptance test*
   The customer, an internal department, or an academic principal tests the almost finished program. The system is checked for compliance with the specification. Thus, it can be part of the development contract [21, p. 7f.]. In certain software development projects this phase can be omitted. If development follows an agile paradigm, acceptance tests are regarded as the *key* test [136].
8. *Pilot test*
   A pilot test can be done with internal departments or with key customers. The almost finished product is installed on a number of systems. It is first tested and eventually can be used productively. Beta tests put the concept of pilot tests to a larger number of testers. They can be conducted in a closed community or public.
9. *Productive usage*
   Software is put into regular operation.

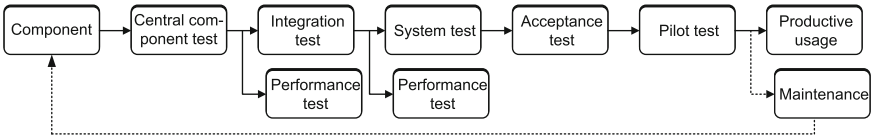

Fig. 2.3 Schematic model of testing phases

10. *Maintenance*
    Optionally, the system is maintained. This is particularly important if development is a periodic process that leads to new releases of the program.

Of course, the above phases can only be realized by companies of sufficient size. Smaller companies and academic developers will not have distinct testers, probably no reserved systems for testing, and a less sophisticated organizational background. In this case, the model gets much simpler and will rely on a skeleton of component, integration, and system test with an optional acceptance test [18, p. 225]. In the opposite direction, a large company could add subphases and even more organizational embedding.

The above model does not behave like the waterfall model [137]. While the order of the phases is fixed—there simply is no reason to do a system test before doing an integration test—there can be feedback in the testing process, it can partly be parallelized, and it can be iterative. This is also reflected in the so called *W-Model*, an extension of the V-Model [138, 139].

The phases are shown in Fig. 2.3. From the left to the right the program under development becomes more mature. At the same time, the technical focus of testing gets lower; acceptance testing is purely functional. If testing is successful, the number of defects found should shrink with each phase. At the same time the severity and with it the costs for removing a defect increases from left to right [133]. Fixing a defect in a finished program could be more than 100 times as costly as avoiding it during specification [6, p. 91f.]. Staging thus rises testing efficiency [21, p. 8f.].

If a software development process model is used, phases might look differently. This applies less to rather formal frameworks such as the Rational Unified Process (RUP) [140] but even more to *agile* models such as Extreme Programming (XP) [136]. Overviews of some widely used models are given by Spillner et al. [24, Chap. 3] and Wallmüller [18, p. 127ff.]. There are ample possibilities for extension. For example, requirements engineering [141] can be combined with testing [142, 143]. This helps to identify requirements, test software based on its specification, and can even help to find defects. Furthermore, it is possible to subdivide phases (e.g. there is the "phase model for low level tests" [22, Chap. 10]). Additions to the phases and their embedding into the overall development process are clearly out of scope of this work.

Agile software development [144, 145] has become increasingly popular. It originates in XP [136], which it is still attributed to [146]. Agile development includes agile testing [147]. It refrains from structuring and formal processes in favor of *agility*. However, agile testing requires management support since it has to be embedded into

a fitting organizational background. Using some agile techniques in combination with formal processes is possible, but in general agile testing is done in alignment with flexible project management approaches such as *Scrum* [148, 149].

2.3.2 Test Driven Development

Usually, testing phases follow development phases. Even if testing and development are parallelized, testing follows coding of a module. Test driven development (TDD) is an approach that changes this basic order. It is also named *test first development* [150, p. 203], which explains its principle. First of all, a test case is written, and then code is implemented. The code's aim is to enable the successful execution of the test case [5, p. 20f.]. Typically, unit testing tools are used [151, p. 36f.]. In general, testing phases still exist and many tests (e.g. static analyses) are done the same way and at the same time as in classical development. However, development always follows the implementation of unit tests.

With test driven development, the implemented functionality tends to be simple. It *only* has to fulfill the tests' requirements. Nevertheless, TDD is said to improve maintainability and to even increase code reusage [150, p. 195]. To work test driven, developers need to write code with much discipline [151, p. 8]. TDD can also decrease development costs because defects are discovered at a very early stage of development [150, p. 202]. They are also likely to be found shortly after they were introduced [151, p. 9] which supports to locate them. However, similar to XP (which it relates to [150, p. 204f.]), TDD is no definite solution. In order to be successfully applied, it has to be in alignment with the philosophy met in a company.

Test driven development has become a field of its own; more detail on it is e.g. given in [150–152].

2.3.3 Documentation

Documenting the testing effort is essential. It is no exaggeration to say that tests that have not been documented are worthless. Only if developer and tester are the same person and defects are fixed immediately, this claim does not hold. Even in this case, testing efficiency is reduced. Companies tend to neglect documenting, though [18, p. 149].

A software test documentation typically tracks the kind of tests conducted, the parameters and conditions that are applied to them, the involved employees, and the outcome. It also varies with the testing phases to reflect changing aims. Thus, a *test plan* could also be attributed to the documentation [4, p. 522]. A universal documentation helps to learn from former test runs. It greatly increases the reusability of test cases and hence improves regression testing [4, p. 517f.]. Moreover, it helps to control testing [18, p. 258ff.]. Data (e.g. on the success) can be aggregated and

checked. This can be extended to become a *test controlling* which helps to proactively avoid testing related problems [153]. With a growing database, the test documentation can also be a resource of knowledge on testing related problems [26, p. 56ff.].

The starting level for a test documentation usually is characterized by the usage of spreadsheets. Preparing such spreadsheets can be seen as a managerial task [21, Chap. 5]; this is especially true if templates become mandatory. Tool support increases efficiency [18, p. 155]. If a dedicated system is employed, it can greatly reduce the manual effort for documenting tests [67]. Aligning this system with a *test case management* system enables automatic reporting of test cases [67]. Testers merely need to check whether the automated data import was successful and whether documented test cases should be annotated with additional information.

Documenting tests can be seen as a strategic decision. It has to be aligned with the preconditions met and the chosen management approach towards testing. Documents can make up a hierarchy that deals with varying levels of technical details [24, Chap. 4]. Documenting tests can become part of a *test concept*. It is even possible to set up test documentation phases.

The test documentation can be used to install a *test reporting*. Reporting helps to learn about a project's status and the already implemented functionality. Furthermore, it can help to determine how the developed program's functionality will look like [6, p. 17]. It might also be used to determine when the project will be finished.

Besides being discussed in the literature on testing, there also is the IEEE Standard 829 (*IEEE Standard for Software Test Documentation*) [154, 155] which gives detailed advice on how to maintain a test documentation. Documentation obligations are also becoming part of IT related business frameworks such as the earlier mentioned CobiT [26, p. 58].

It has also been suggested that the documentation of the software under development should be tested [6, Chap. 20]. However, it is questionable whether this procedure satisfies the characteristics of being a test. In addition, technical details are less important for software documentation papers such as handbooks or manuals. Checking the documentation will require the collaboration of software developers, GUI designers, and writers or editors that prepare comprehensible texts. In conclusion, it is agreed with those authors who distinguish between testing and checking (cf. e.g. [5, Chap. 6]) and who regard an assessment of program documentations as *checking*. Nevertheless, documenting is essential in any phase of a software development process that strives for software quality [18, p. 149f.].

2.3.4 Test Management

If testing is done in teams, software developing companies employ an at least high two-digit number of employees, or companies emphasize a controlled testing process, test management becomes important. Test management therefore can be seen as a *management discipline* with strategic impact. Of course, even for single developers

it is important to *manage* their testing effort and for example work phase oriented (see Sect. 2.3.1); this section deals with the management of testing in companies.

Test management is particularly advisable due to the high level of specialization among testers [18, p. 240]. There is a number of roles that need knowledge of test management:

- Testers need to understand how to work in teams and in alignment with software developers. In general, developers should not test the program they have implemented; testers should not try to fix defects they find [96, p. 258]. Thus, close cooperation is needed.
- For large projects, test managers can be installed that take similar functions like development project managers or senior software architects. In particular, they should set up *testing plans* [96, p. 273].
- For projects with a three-digit number of involved employees, a central testing manager might coordinate teams of testers.
- Project managers and controllers need to understand how testing is done.
- Executives should understand the impact of testing as well as its preconditions. Otherwise, testing is likely to be neglected instead of being seen as an important task (cf. "culture of testing" [153]).

Traditionally, test management has been neglected or even been seen as superfluous [18, p. 242f.]. It, however, can contribute to improved software development processes [18, p. 240ff.]. Thus, it supports to decrease costs while increasing software quality. Consequently, modern textbooks on testing often have at least one chapter on test management. In can be aligned with a holistic quality management approach (cf. Sect. 2.1.4.4 on the DIN EN ISO 9001 standard).

According to Wallmüller, software quality management should follow a set of principles [18, p. 17ff.]:

- It should define operational properties to measure quality. Furthermore, it should support to survey and to analyze them. In this context, it is particularly important to generate *metrics* and infer advisable actions from them. Metrics are key figures that measure technical or individual performance [24, p. 283ff.].
- It should provide product oriented and project oriented quality planning.
- Feedback from quality assessment should be used to improve processes. Ideally, this would lead to a process of continuous refinements [26, p. 33].
- Quality assessment should always be conducted by at least two employees.
- It should develop a proactive quality control. The maxim is to avoid defects in software products rather than to find failures and fix the corresponding defects.
- It should support testing and development with the aim of finding failures in early phases of development.
- Integrated quality management should be provided. This means that test management must not be a task of its own but interwoven with the development process.
- It should support independent quality assessments. In particular, it should ensure that software is not only tested by its developers to prevent conflict of aims.
- It should examine the effort for quality management and seek to improve it.

At the same time, unsuccessful strategies should be avoided. In the style of anti patterns in software development (Sect. 2.2.1.2, p. 45) they are called *software project management anti-patterns* and collected in the literature [156, 157].

In contrast to other disciplines that have a technical background and require management, test managers require profound knowledge of the technology [21]. Therefore, test management does not only consist of tasks that can be derived from the above mentioned principles. Besides this focus on supporting the strive for high quality software and continuous process improvements, it should also provide technical support. This can be done in the form of infrastructure support. A *test lab*—also known as a *test center* [26, p. 47ff.]—provides the technical infrastructure for timely tests under realistic conditions [21, p. 293ff.]. Test centers can be part of a sophisticated *testing organization* [18, p. 255f.]. Furthermore, test management deals with staffing a test team, test outsourcing, test project management, test planning [18, p. 242ff.], and the economics of testing [21, Chaps. 8, 10, 12]. Finally, test managers are also responsible for the education of the testing staff and personnel that does preparatory work for testers [24, Chap. 10].

A detailed overview of test management and software quality management is given by Black [21], Spillner et al. [24] and Wallmüller [18].

2.3.5 Certification

Certificates in the context of the IT industry attest humans that they have a certain level of knowledge or skill in an IT related field. They are very common in IT jobs and are seen as an additional form of education [158, 159]. Apparently, certificates have been used in the industry for more than three decades [160]. Certification is particularly common in the fields of enterprise applications (cf. [161–163]) and programming languages (cf. [164–166]). In addition to certification for humans, it also is possible to certify companies' adherence to standards.

After scanning the literature, it can be said that testing is particularly popular for certifications. There is a myriad of textbooks on testing related certifications (for instance [167]) and especially on certifications by the International Software Testing Qualifications Board (ISTQB) (such as [168, 169]). There are not only basic, but also advanced courses that usually require a basic course as their foundation [170]. A Google web search [171] shows a great number of courses to prepare for certificates on testing. It also shows that the ISTQB certification seems to be the most popular one; nevertheless, there are several other courses such as the certified software tester (CSTE) [172] or the Standardisation Testing and Quality Certification (STQC) [173]. Unfortunately, there are no scientifically assessed numbers available on the total number of certified testers per program; numbers found on the Web vary widely.

Interestingly, general books on software testing such as [2, 4] do not cover certificates. It is however dealt with in newer literature on test management [21, p. 334ff.].

Certifications "might or might not" involve training, but each certified person has passed an exam [21, p. 334]. Thus, without importance whether the person has taken

a course and expanded his knowledge or not, companies can rely on the certified knowledge. In other words: this person is very likely to master a predefined level of knowledge and has a set of testing skills. Nevertheless, it can be argued that certifications do not reflect experience and so called *soft skills*. Both can be seen as very important for testing [26, p. 48]. Besides, certifications are not free of charge. Typical courses cost a low four-digit sum plus a high three-digit sum for the exam (cf. [174]). If organized as on-the-job training and targeting employees that do not yet have testing skills, this fee should be acceptable for companies. However, it is rather high if payed by individuals before applying for a job. Notwithstanding, a discussion of pros and cons of testing certificates from a business or from an employee's perspective is out of scope of this work. A decision for or against certifications is particularly bound to a company's culture [26, p. 58ff.].

2.4 Tool Support and Automatization

Textbooks on software testing usually advise the use of test tools and test automatization. This is common sense: test tools promise to make testing much more effective by offering techniques humans are incapable of; automatization promises to relieve humans of repetitive tasks and make testing more economic [101, p. 159]. However, tools are much less commonly used than usually expected. Automatization is hardly used at all [26, p. 17f.]. These findings from the IHK project only seemingly do not align with the literature. None of the textbooks cited in this chapter suggest specific tools (besides very basic solutions) or give advice on automatization that is easy to implement.

Tool support for testing is a topic of its own. Therefore, the current status is introduced only briefly. Test automatization will be given more emphasis in Chap. 4, as the tool Muggl means to automate testing. It has to be noted that most tools not only extend the testing scope but also offer more or less sophisticated automatization. Therefore, this work distinguishes between automatizing some testing tasks which makes testing more convenient, and *true* automatization. The latter describes (almost) completely automated testing. This book does not follow an approach which treats *test tool usage* and *test automatization* as synonymous. This is e.g. assumed by Cleff [5, Chap. 8] and Thaller [7, Chap. 8].

2.4.1 Test Tools

Test tools—also known as computer aided software testing (CAST)—can serve a variety of purposes. In general, they are "vehicles" for test techniques [6, Chap. 5]. Spillner et al. propose the following distinction [24, p. 324ff.]:

- Tools for the management and control of testing have a less technical focus. They are used to plan testing, measure the testing effort and results, and manage the suite of test cases.
- Test case generators, as the name suggests, generate test cases. Generation strategies vary widely as does the level of automatization.
- Analytic tools automate static tests or enable static testing techniques that would not be feasible without tool support.
- Execution of tests is supported by a number of specialized tools. They can work on a low abstraction level and execute suits of test cases or on a high abstraction level and offer *Capture&Replay* capabilities. Examples for the first category are unit test tools such as JUnit [175] or CppUnit [176]. An introduction to unit test tools is given by Vigenschow [101, p. 160ff.]. The second category comprises of tools that particularly focus on GUI testing. They can monitor testers' actions and automatically repeat them at a later time. This can be even done in a script based manner to parameterize such tests [101, p. 191ff.].
- Specialized tools support particular situations. Examples are tools to tests Web applications such as HtmlUnit [177] or Selenium [178] and mutation test tools.

Much more detailed classifications on a more technical level have been suggested [179]. Computer aided software engineering (CASE) and similar tools can also be regarded as a way to improve testing [18, p. 169f.]. However, a very detailed view is unsuitable for an overview as given here. Another suggestion is to align the classification with that of testing techniques. Consequently, dynamic, static, formal, and analyzing tools can be distinguished [2, Chap. 12]. Bath and McKay use a combination of the latter and Spillner et al.'s approach [3].

Test tools are particularly useful if they are capable of taking over repetitive tasks [5, Chap. 8] [21, p. 81ff.], if a high level of precision is needed, if automated metric generation and documenting of tests is desired [2, p. 392], or if large amounts of data need to be processed. The main aim thus is a higher efficacy [3, p. 324f.]. A checklist to judge tools by usefulness is given by Liggesmeyer [2, p. 404f.]. While it is undisputed that test tools can highly increase testing effectiveness and efficiency, there are a number of risks. They arise from unrealistic expectations towards test tools, underestimating effort and time required to introduce tools, putting too little effort into continuous improvement and maintenance of tools, and unreflected trust in tools [5, p. 272ff.]. Another important decision is *make-or-buy*; merits, costs, and risks have to be taken into account [3, p. 349f.]. Standard solutions usually are powerful but costly and probably inflexible; implementing tools requires much time but allows to tailor them to processes [26, p. 30f.].

Evaluating test tools requires considerable effort. It is not merely a decision that weights price and performance but has to take into account the effect on processes as well as other organizational consequences, the target adequateness [6, p. 188ff.] of the tools, and other side-effects [24, p. 330ff.]. Ample time to install a tool and get into working with it should be considered [7, p. 233]. Tools should also be selected based on their integration capabilities [3, p. 340ff.] [180]. Introducing selected tools should be controlled and the merits of the introduction continuously checked

[5, p. 275ff.] [6, Chap. 5] [24, p. 330ff.]. A guideline for choosing and installing tools is e.g. provided by Perry [6, Chap. 5]. Initial experiences from improvements due to the usage of tests tools are presented by Thaller [7, p. 239ff.]. After all, choosing tools should be based on a process that evaluates the usefulness of test automatization [4, p. 497].

2.4.2 True Automatization

Defining the ideal testing technique is no challenge. Consider a tool that requires source code or a program file as input and that generates a suite of test cases that detects all possible failures. It furthermore helps to suggest which defects cause the failures and aids in fixing them. Unfortunately, no such tool exists. It is highly questionable whether it is possible to implement such a tool at all. As stated at the beginning of this chapter, it is only possible to test against a specification. Therefore, true automatization that does not require human intervention requires a computer-readable specification. This is for example done by model checking. However, the effort is high and impractical in many cases. Merely repeating steps does not qualify as automatization. It goes beyond what Capture and Replay tools do [181, p. 26]. Nevertheless, automatization is an important aim. The maxim is to relief humans of as much effort as possible.

Automatization is recommended for testing programs of considerable size [6, p. 72f.]. Regrettably, automatization becomes more complicated with a program's size. In the course of developing Muggl, we noticed that it is notably easier to generate test cases for single algorithms than for modules or even extensive programs.

It is important to note that automatization is no universal cure [3, p. 343ff.] unless it is *perfect* or reasonably intelligent. Self-verification of software could be impossible[7] and it is questionable whether computer systems will soon reach a level of artificial intelligence (AI) that would allow them to replace human testers in manual testing task (cf. with the discussion of the *AI winter* [182, 183]). Automatization is simple and feasible in some cases but desirable yet impossible in others [4, p. 497ff.]. It should be based on an analysis of usefulness and costs [184]. Automatization requires a highly structured approach and much discipline. Or, as Bath and McKay put it [3, Chap. 18]: "Automatizing chaos merely accelerates it."[8]

In general, four fields of testing are suitable for extended automatization:

- The generation of metrics [4, p. 502ff.]. Tools that analyze source code, bytecode or programs and calculate key figures are widely available [26, Appendix B].
- Automatically generating the test documentation while executing tests poses little technical challenge [67].

[7] As of today, it is seen as impossible due to the *halting problem*. Programs might verify themselves under certain conditions, but no universal approach is known.

[8] Translated from the German original by the author.

- Tools for static analysis and automated proofs are currently in development but very promising [4, p. 505ff.].
- TCG is possible in a variety of steps from manual to fully automated.

A problem of test automatization is that it is "more expensive to automate a test than to perform it once manually" [181, p. 5]. Consequently, it has to be well planned in order to avoid a waste of resources. Nevertheless, automatization offers a number of merits [181, p. 9f.]:

- It helps to run existing tests on new versions of a program. Also, the reuse of test cases can be improved.
- It enables to run regression tests more often—ideally, they are repeated continuously.
- Tests can be performed that would be impossible without (half-)automated tool support.
- Resources can be utilized more efficiently if tasks are automated while humans focus on techniques that require supervision.
- Economic and organizational targets such as an earlier time-to-market and a higher confidence in products can be met.
- Testing might become more reliable [22, p. 472].
- Testing of derivative programs, upgrades, etc. becomes easier [7, p. 228f.].

Automatization comes at a price; it is prone to cause problems if not implemented carefully. Despite its merits it has several limitations [181, p.10ff.]:

- Management expectations often are unrealistic. Automatizing takes much time and effort until satisfying results can be reached. At the same time it is almost impossible to introduce test automatization without management support.
- Automatization does not necessarily find additional defects. Automatization *might* find more defects, but the aim could also be to find defects more efficiently.
- Maintenance of automated tests and test automatization tools can require a considerable amount of time and tie up resources. Unfortunately, interoperability of commercial tools is a problem as well [26, p. 87ff.].
- Automatization does not replace manual testing. Despite progress in automatization, manual tests will be required for the foreseeable future. If an extreme level of reliability is required, manual testing with tremendous effort is likely to reveal more failures than automated testing (also cf. [26, p. 17f.]). In particular, tools neither have "imagination" [181, p. 24] nor intuition.

It is, nonetheless, refrained from supporting the argument that "test automatization does not improve effectiveness" [181, p. 24]. While in many cases manual testing with high effort is more effective and automated testing might "adversely affect the evolvability of testing" [181, p. 24], progress with Muggl is promising. Automated testing is not a "quick cure" [185, p. 350]. But under certain—admittedly still limited—conditions, automated test case generation is effective (also see Chaps. 7 and 8).

References

1. Myers, G.J., Sandler, C.: The Art of Software Testing, 2nd edn. Wiley, Chichester (2004)
2. Liggesmeyer, P.: Software-Qualität: Testen, Analysieren und Verifizieren von Software, 2nd edn. Spektrum-Akademischer Verlag, Berlin (2009)
3. Bath, G., McKay, J.: Praxiswissen Softwaretest—Test Analyst und Technical Test Analyst. Dpunkt, Heidelberg (2010)
4. Pezze, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2007)
5. Cleff, T.: Basiswissen Testen von Software. W3L GmbH (2010)
6. Perry, W.E.: Software Testen. mitp, Bonn (2003)
7. Thaller, G.E.: Software-Test, 2nd edn. Heise, Hannover (2002)
8. Rubin, J., Chisnell, D.: Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests. Wiley, Hoboken (2008)
9. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers, San Francisco (1994)
10. Leventhal, L., Barnes, J.: Usability Engineering: Process, Products and Examples. Prentice Hall, Upper Saddle River (2007)
11. Sears, A., Jacko, J.A. (eds.): The Human–Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications, 2nd edn. CRC Press, Boca Raton (2008)
12. Sharp, H., Rogers, Y., Preece, J.: Interaction Design: Beyond Human–Computer Interaction, 2nd edn. Wiley, Hoboken (2007)
13. Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S.: Designing the User Interface: Strategies for Effective Human–Computer Interaction, 5th edn. Addison Wesley, Reading (2009)
14. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2006)
15. IEEE, The Institute of Electrical and Electronics Engineers, Inc.: IEEE Std 1044-2009: IEEE Standard Classification for Software Anomalies. New York (2010)
16. Marick, B.: The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing. Prentice-Hall, Upper Saddle River (1995)
17. Meyer, B.: Seven principles of software testing. Computer **41**(8), 99–101 (2008)
18. Wallmüller, E.: Software-Qualitätsmanagement in der Praxis, 2nd edn. Hanser, München (2001)
19. IEEE, The Institute of Electrical and Electronics Engineers, Inc.: IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology. New York (1990)
20. IEEE, The Institute of Electrical and Electronics Engineers, Inc.: IEEE Std 610.12 (H)-1990: IEEE Standard Glossary of Software Engineering Terminology (HyperCard Stack). New York (1990)
21. Black, R.: Managing the Testing Process, 3rd edn. Wiley, Indianapolis (2009)
22. Pol, M., Koomen, T., Spillner, A.: Management und Optimierung des Testprozesses: Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap, 2nd edn. dpunkt, Heidelberg (2002)
23. Myers, G.J., Sandler, C.: The Art of Software Testing. Wiley, Chichester (2004)
24. Spillner, A., Roßner, T., Winter, M., Linz, T.: Praxiswissen Software Test—Test Management. Dpunkt, Heidelberg (2008)
25. Roitzsch, E.H.P.: Analytische Softwarequalitätssicherung in Theorie und Praxis: Der Weg zur Software mit hoher Qualität durch statisches Prüfen, dynamisches Testen, formales Beweisen. Monsenstein und Vannerdat (2005)
26. Majchrzak, T.A., Kuchen, H.: IHK-Projekt Softwaretests: Auswertung. In: Working Papers, Vol. 2. Förderkreis der Angewandten Informatik an der Westfälischen Wilhelms-Universität, Münster e.V. (2010)
27. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking. In: Proceedings of the NATO Advanced Study Institute on Deductive Program Design, pp. 305–349. Springer, Secaucus (1996)

28. Baier, C., Katoen, J.P.: Principles of Model Checking Representation and Mind Series. The MIT Press, Cambridge (2008)
29. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph., McKenzie, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer, Heidelberg (2001)
30. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2008)
31. ISO: ISO/IEC 12119:1994 Information Technology—Software Packages—Quality Requirements and Testing. Geneva (1994)
32. http://www.istqb.org/
33. http://www.german-testing-board.info/
34. http://www.german-testing-board.info/de/lehrplaene.shtm
35. Spillner, A., Linz, T.: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester—Foundation Level nach ISTQB-Standard. dpunkt (2010)
36. http://www.w3.org/
37. Abran, A., Bourque, P., Dupuis, R., Moore, J.W., Tripp, L.L. (eds.): Guide to the Software Engineering Body of Knowledge—SWEBOK. IEEE Press, Piscataway (2004)
38. Höhn, R., Höppner, S. (eds.): Das V-Modell XT: Grundlagen, Methodik und Anwendungen. Springer, Berlin (2008)
39. http://www.v-modell-xt.de/
40. Dröschel, W., Heuser, W., Midderhoff, R. (eds.): Inkrementelle und Objektorientierte Vorgehensweisen mit dem V-Modell 97. Oldenbourg, München (1998)
41. n. A.: DO-178B, Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA) (1992)
42. CENELEC: DIN EN 50128, Railway applications. Communications, signalling and processing systems. Software for railway control and protection systems. European Committee for Electrotechnical Standardization (CENELEC) (2001)
43. IEC: IEC 62304:2006 Medical Device Software—Software Life Cycle Processes. Geneva (2006)
44. IEC: IEC 61508 Edition 2.0. International Electrotechnical Commission (IEC) (2010)
45. n. A.: The Official Introduction to the ITIL 3 Service Lifecycle: Office of Government Commerce. The Stationery Office (2007)
46. http://www.isaca.org/Knowledge-Center/cobit/Documents/CobiT-4.1-Brochure.pdf
47. n. A.: CMMI for Development, Version 1.2. Carnegie Mellon Software Engineering Institute (2006)
48. Finkelstein, A.: A software process immaturity model. SIGSOFT Softw. Eng. Notes **17**(4), 22–23 (1992)
49. http://www.grisha.ru/cmm/cimm.htm
50. Pol, M., Teunissen, R., van Veenendaal, E.: Gestructureerd testen: Een inroductie tot Tmap. Tutein Nolthenius (1998)
51. Pol, M., Teunissen, R., van Veenendaal, E.: Software Testing: A Guide to the TMap Approach. Addison-Wesley, Boston (2001)
52. Koomen, T., Pol, M.: Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing. Addison-Wesley, Boston (1999)
53. Sogeti: TPI Next—Business Driven Test Process Improvement. UTN Publishers, The Netherlands (2009)
54. ISO: ISO 9000:2005 Quality Management Systems—Fundamentals and Vocabulary. Geneva (2005)
55. ISO: ISO 9001:2008 Quality Management Systems—Requirements. Geneva (2008)
56. ISO: ISO 9004:2009 Managing for the Sustained Success of an Organization—a Quality Management Approach. Geneva (2009)
57. Kanji, G.K., Asher, M.: 100 Methods for Total Quality Management. Sage Publications, London (1996)

58. Balzert, H.: Lehrbuch der Softwaretechnik: Softwaremanagement, 2nd edn. Spektrum Akademischer Verlag, Heidelberg (2008)
59. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: a successful application of B in a large project. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM'99—Formal Methods, Lecture Notes in Computer Science, vol. 1708, p. 712. Springer, Heidelberg (1999)
60. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models. In: Proceedings of the 2nd World Congress on Formal Methods, FM '09, pp. 708–723. Springer, Berlin (2009)
61. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
62. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**, 271–281 (1972)
63. Floyd, R.W.: Assigning meanings to programs. Proc. Symp. Appl. Math. **19**, 19–32 (1967)
64. Zilles, S.N.: Algebraic Specification of Data Types. Project Mac Progress Report 11. Massachusetts Institute of Technology, Cambridge (1974)
65. Bergstra, J.A., Heering, J., Klint, P.: Algebraic Specification. ACM, New York (1989)
66. Sneed, H.M., Winter, M.: Testen Objektorientierter Software. Hanser, München (2002)
67. Majchrzak, T.A.: Improving the technical aspects of software testing in enterprises. Int. J. Adv. Comput. Sci. Appl. **1**(4), 1–10 (2010)
68. Oracle: How to write Doc comments for the Javadoc tool. http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html (2004)
69. Sutter, H., Alexandrescu, A.: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley, Boston (2004)
70. Kellerwessel, H.: Programmierrichtlinien in der Praxis. mitp, Bonn (2002)
71. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, pp. 439–449. IEEE Press, Piscataway (1981)
72. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
73. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. **29**(3), 155–163 (1988)
74. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)
75. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Syst. J. **15**(3), 182–211 (1976)
76. Frühauf, K., Ludewig, J., Sandmayr, H.: Software-Prüfung. vdf Hoschulverlag, Zürich (1991)
77. Black, R.: Pragmatic Software Testing. Wiley, Indianapolis (2007)
78. Koenig, A.: Patterns and antipatterns. JOOP **8**(1), 46–48 (1995)
79. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addision-Wesley, München (1995)
80. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)
81. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
82. Opdyke, W.F., Johnson, R.E.: Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In: Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPA) (1990)
83. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education, London (2004)
84. Majchrzak, T.A., Kuchen, H.: Automated test case generation based on coverage analysis. In: TASE '09: Proceedings of the 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 259–266. IEEE Computer Society, Washington, DC (2009)
85. Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. SIGPLAN Notices **8**(8), 12–26 (1973)
86. Hennell, M.A., Woodward, M.R., Hedley, D.: On program analysis. Inf. Process. Lett. **5**(5), 136–140 (1976)

87. Woodward, M.R., Hennell, M.A.: On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC. Inf. Softw. Technol. **48**(7), 433–440 (2006)
88. Alper, M.: Professionelle Softwaretests: Praxis der Qualitätsoptimierung Kommerzieller Software. Vieweg, Braunschweig (1994)
89. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Boston (2005)
90. Watkins, J.: Testing IT: An Off-the-Shelf Software Testing Process. Cambridge University Press, New York (2001)
91. Lazic, L., Mastorakis, N.: Orthogonal array application for optimal combination of software defect detection techniques choices. W. Trans. Comp. **7**(8), 1319–1336 (2008)
92. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**(6), 418–421 (2004)
93. Richardson, D.J., Clarke, L.A.: Partition analysis: a method combining testing and verification. IEEE Trans. Softw. Eng. **11**(12), 1477–1490 (1985)
94. Richardson, D.J., Clarke, L.A.: A partition analysis method to increase program reliability. In: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, pp. 244–253. IEEE Press, Piscataway (1981)
95. Podgurski, A., Yang, C.: Partition testing, stratified sampling, and cluster analysis. SIGSOFT Softw. Eng. Notes **18**(5), 169–181 (1993)
96. Mayr, H.: Projekt Engineering: Ingenieurmäßige Softwareentwicklung in Projektgruppen, 2nd edn. Hanser, München (2005)
97. Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, New York (1996)
98. Sankaran, N.: A bibliography on garbage collection and related topics. SIGPLAN Notices **29**(9), 149–158 (1994)
99. Hofstedt, P., Wolf, A.: Einführung in die Constraint-Programmierung, Grundlagen, Methoden, Sprachen, Anwendungen. Springer, Heidelberg (2007)
100. Eyre-Todd, R.A.: The detection of dangling references in C++ programs. ACM Lett. Program. Lang. Syst. **2**(1–4), 127–134 (1993)
101. Vigenschow, U.: Objektorientiertes Testen und Testautomatisierung in der Praxis. Konzepte, Techniken und Verfahren. Dpunkt, Heidelberg (2005)
102. Georgieva, K.: Conducting FMEA over the software development process. SIGSOFT Softw. Eng. Notes **35**(3), 1–5 (2010)
103. Scientific N., Program T.I. (2000) Failure Modes and Effects Analysis (FMEA)—A Bibliography. Technical Report, NASA Langley Technical Report Server
104. Neufelder, A.M.: Ensuring Software Reliability. Marcel Dekker, New York (1992)
105. Vouk, M.A.: Back-to-back testing. Inf. Softw. Technol. **32**(1), 34–45 (1990)
106. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Program Mutation: A New Approach to Program Testing. Software Testing pp. 107–128 (1979)
107. Howden, W.E.: Weak mutation testing and completeness of test sets. IEEE Trans. Softw. Eng. **8**(4), 371–379 (1982)
108. Howden, W.E.: Completeness criteria for testing elementary program functions. In: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, pp. 235–243. IEEE Press, Piscataway (1981)
109. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: SIGPLAN Notices, pp. 268–279. ACM Press (2000)
110. Paris, J., Arts, T.: Automatic testing of TCP/IP implementations using QuickCheck. In: ERLANG '09: Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG, pp. 83–92. ACM, New York (2009). http://doi.acm.org/10.1145/1596600.1596612
111. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with QuickCheck and PULSE. In: ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, pp. 149–160. ACM, New York (2009)
112. http://quickcheck.dev.java.net/

113. http://www.jcheck.org/
114. http://software.legiasoft.com/quickcheck/
115. http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html
116. Rumbaugh, J.E., Blaha, M.R., Premerlani, W.J., Eddy, F., Lorensen, W.E.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (1991)
117. Jones, C.: The Economics of Object-Oriented Software. Technical Report, Software Productivity Research, Massachusetts (1997)
118. Beizer, B.: Software System Testing and Quality Assurance. Van Nostrand Reinhold, New York (1984)
119. Dybjer, P., Haiyan, Q., Takeyama, M.: Verifying Haskell programs by combining testing and proving. In: Proceedings of the Third International Conference on Quality Software, QSIC '03, pp. 272–279. IEEE Computer Society, Washington, DC (2003)
120. Lu, L., Greenfield, P.: Logic program testing based on abstract interpretation. In: Proceedings of the International Conference on Formal Methods in Programming and Their Applications, pp. 170–180. Springer, London (1993)
121. Silva, J., Vidal, G.: Forward slicing of functional logic programs by partial evaluation. Theory Pract. Log. Program. **7**(1–2), 215–247 (2007)
122. Fischer, S., Kuchen, H.: Data-flow testing of declarative programs. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 201–212. ACM (2008)
123. Klein, T.: Buffer Overflows und Format-String-Schwachstellen. Dpunkt, Heidelberg (2004)
124. Poguntke, W.: Basiswissen IT-Sicherheit: Das wichtigste für den Schutz von Systemen & Daten. W3L GmbH (2007)
125. Wysopal, C., Nelson, L., Zovi, D.D., Dustin, E.: The Art of Software Security Testing: Identifying Software Security Flaws. Addison-Wesley, Boston (2006)
126. Hope, P., Walther, B.: Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast. O'Reilly, Beijing (2008)
127. Andrews, M., Whittaker, J.A.: How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. Addison-Wesley, Boston (2006)
128. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/itgrundschutz_node.html
129. ISO: ISO/IEC 9126:2001 Software engineering—Product quality. Geneva (2001)
130. Grood, D.: Test Goal: Result-Driven Testing. Springer, Heidelberg (2008)
131. Forsberg, K., Mooz, H.: The relationship of systems engineering to the project cycle. Eng. Manag. J. **4**(3), 36–38 (1992)
132. Majchrzak, T.A.: Status quo of software testing—regional findings and global inductions. In: Proceedings of the 9th ISOneWorld Conference. The Information Institute (2010)
133. Majchrzak, T.A.: Status quo of software testing—regional findings and global inductions. J. Inf. Sci. Technol. **7**(2) (2010)
134. Thomas, D., Hunt, A.: Mock objects. IEEE Softw. **19**(3), 22–24 (2002)
135. Sneed, H.M., Baumgartner, M., Seidl, R.: Der Systemtest, 2nd edn. Hanser, München (2009)
136. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston (1999)
137. Royce, W.W.: Managing the development of large software systems: concepts and techniques. In: ICSE '87: Proceedings of the 9th International Conference on Software Engineering, pp. 328–338. IEEE Computer Society Press, Los Alamitos (1987)
138. Spillner, A.: Das W-Modell—Testen als paralleler Prozess zum Software-Entwicklungsprozess. Softwaretechnik-Trends **21**(1), 4–5 (2001)
139. Spillner, A.: Management des Testprozesses von Anfang an—das W-Modell. Lect. Notes Inform. **P-23**, 65–76 (2002)
140. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, Boston (2003)
141. Sommerville, I., Kotonya, G.: Requirements Engineering: Processes and Techniques. Wiley, New York (1998)
142. Lim, M., Sadeghipour, S.: Werkzeugunterstützte Verknüpfung von anforderungen und tests—voraussetzung für eine systematische qualitätssicherung. Softwaretechnik-Trends **28**(3), 32–33 (2008)

143. Tochtrop, G.: Testexperte als Anforderungsmanager—Ein Erfahrungsbericht. Softwaretechnik-Trends **28**(3), 34–37 (2008)
144. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Upper Saddle River (2003)
145. Shore, J., Warden, S.: The Art of Agile Development. O'Reilly, Beijing (2007)
146. Warden, S., Shore, J.: The Art of Agile Development: With Extreme Programming. O'Reilly Media, Beijing (2007)
147. Adzic, G.: Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing. Neuri Limited, London (2009)
148. Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley, Boston (2009)
149. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press, Redmond (2004)
150. Beck, K.: Test-Driven Development by Example. Addison-Wesley, Boston (2002)
151. Koskela, L.: Test driven: practical tdd and acceptance tdd for java developers. Manning Publications, Greenwich (2007)
152. Astels, D.: Test Driven Development: A Practical Guide. Prentice Hall, Upper Saddle River (2003)
153. Majchrzak, T.A.: Best practices for the organizational implementation of software testing. In: Proceedings of the 43rd Annual Hawaii International Conference on System Sciences (HICSS-43), pp. 1–10. IEEE Computer Society (2010)
154. IEEE, The Institute of Electrical and Electronics Engineers, Inc.: IEEE Std 829-1998: IEEE Standard for Software Test Documentation. New York (1998)
155. Konradt, U., Sarges, W. (eds.): E-Recruitment und E-Assessment: Rekrutierung, Auswahl und Beratung von Personal im Inter- und Intranet. Hogrefe, Göttingen (2003)
156. Stamelos, I.: Software project management anti-patterns. J. Syst. Softw. **83**(1), 52–59 (2010)
157. Brown, W.J., III, H.W.S.M., Thomas, S.W.: Anti Patterns in Project Management. Wiley, New York (2000)
158. Hitchcock, L.: Industry certification: value, validity, and a place for SoDIS. SIGCSE Bull. **37**(4), 59–63 (2005)
159. Hitchcock, L.: Industry certification and academic degrees: complementary, or poles apart? In: Proceedings of the 2007 ACM SIGMIS CPR Conference on Computer Personnel Research: The Global Information Technology Workforce, SIGMIS CPR '07, pp. 95–100. ACM, New York (2007)
160. Ketchel, J.S.: Certification for the computer professional—individual reparation strategies. In: Proceedings of the ACM '81 Conference, ACM '81, pp. 234–238. ACM, New York (1981)
161. Lawson, S.: DB2(R) for z/OS(R) Version 8 DBA Certification Guide. IBM Press (2004)
162. Stanek, W.: Microsoft Vista Certification Exams in a Nutshell. O'Reilly Media, Beijing (2007)
163. Redbooks, I.: IBM Tivoli Monitoring V 6.1. Vervante (2006)
164. Sybex Inc.: Java 2 Certification Virtual Test Center. SYBEX Inc., Alameda (2001)
165. Manning, W.: SCJP Sun Certified Programmer for Java 6 Exam. Emereo Pty Ltd, London (2009)
166. Coggeshall, J., Tabini, M.: The Zend PHP Certification Practice Test Book—Practice Questions for the Zend Certified Engineer Exam. Marco Tabini & Associates (2005)
167. Shuja, A., Krebs, J.: IBM Rational Unified Process Reference and Certification Guide: Solution Designer. IBM Press (2007)
168. Graham, D., Veenendaal, E.V., Evans, I., Black, R.: Foundations of Software Testing: ISTQB Certification. International Thomson Business Press (2008)
169. Bradtke, R.: ISTQB 100 success Secrets—ISTQB Foundation Certification Software Testing the ISTQB Certified Software Tester 100 Most Asked Questions. Emereo Pty Ltd, London (2008)
170. Black, R.: Advanced Software Testing—Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst. Rocky Nook (2008)
171. http://www.google.de/search?q=testing+certification
172. http://www.softwarecertifications.org/qai_cste.htm

173. http://www.stqc.nic.in/
174. http://knowledge-department.de/seminare.html
175. http://www.junit.org/
176. http://sourceforge.net/projects/cppunit/
177. http://htmlunit.sourceforge.net/
178. http://seleniumhq.org/
179. Brössler, P., Siedersleben, J.: Softwaretechnik: Praxiswissen für Softwareingenieure. Hanser, München (2000)
180. Majchrzak T.A.: Technische aspekte des erfolgreichen testens von software in unternehmen. In: Knoop, J., Prantl, A. (eds.) Schriftenreihe des Instituts für Computersprachen, Bericht 2009-X-1: 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung, pp. 193–207. Technische Universität Wien, Maria Taferl (2009)
181. Fewster, M., Graham, D.: Software Test Automation: Effective Use of Test Execution Tools. ACM Press, New York (1999)
182. Crevier, D.: AI: The Tumultuous History of the Search for Artificial Intelligence. Basic Books, New York (1994)
183. Hendler, J.: Avoiding another AI winter. IEEE Intell. Syst. **23**(2), 2–4 (2008)
184. Menzel, M.: Software-Testautomatisierung: Leitfaden für die effiziente Einführung. Vdm Verlag Dr. Müller (2006)
185. Bates, C.: Experiences with test automation. In: Fewster, M., Graham, D. (eds.) Software Test Automation: Effective Use of Test Execution Tools, ACM Press, New York (1999)

http://www.springer.com/978-3-642-27463-3

Improving Software Testing
Technical and Organizational Developments
Majchrzak, T.A.
2012, XVIII, 160 p. 17 illus., Softcover
ISBN: 978-3-642-27463-3