Contents

1 Introduction 1

2 Fundamentals of Testing 5

2.1 Terms and Motivation ... 6

2.1.1 Error, Defect, and Bug Terminology ... 7

2.1.2 Testing Terms ... 8

2.1.3 Software Quality ... 11

2.1.4 Test Effort ... 13

2.2 The Fundamental Test Process ... 17

2.2.1 Test Planning and Control ... 19

2.2.2 Test Analysis and Design ... 22

2.2.3 Test Implementation and Execution ... 25

2.2.4 Test Evaluation and Reporting ... 28

2.2.5 Test Closure Activities ... 30

2.3 The Psychology of Testing ... 31

2.4 General Principles of Testing ... 33

2.5 Ethical Guidelines ... 35

2.6 Summary ... 36

3 Testing in the Software Life Cycle 39

3.1 The General V-Model ... 39

3.2 Component Test ... 42

3.2.1 Explanation of Terms ... 42

3.2.2 Test objects ... 43
3.2.3 Test Environment ... 43
3.2.4 Test objectives ... 46
3.2.5 Test Strategy ... 48
3.3 Integration Test ... 50
3.3.1 Explanation of Terms ... 50
3.3.2 Test objects ... 52
3.3.3 The Test Environment ... 53
3.3.4 Test objectives ... 53
3.3.5 Integration Strategies ... 55
3.4 System Test ... 58
3.4.1 Explanation of Terms ... 58
3.4.2 Test Objects and Test Environment ... 59
3.4.3 Test Objectives ... 60
3.4.4 Problems in System Test Practice ... 60
3.5 Acceptance Test ... 61
3.5.1 Contract Acceptance Testing ... 62
3.5.2 Testing for User Acceptance ... 63
3.5.3 Operational (Acceptance) Testing ... 64
3.5.4 Field Testing ... 64
3.6 Testing New Product Versions ... 65
3.6.1 Software Maintenance ... 65
3.6.2 Testing after Further Development ... 67
3.6.3 Testing in Incremental Development ... 68
3.7 Generic Types of Testing ... 69
3.7.1 Functional Testing ... 70
3.7.2 Nonfunctional Testing ... 72
3.7.3 Testing of Software Structure ... 74
3.7.4 Testing Related to Changes and Regression Testing ... 75
3.8 Summary ... 76

4 Static Test 79

4.1 Structured Group Evaluations ... 79

4.1.1 Foundations ... 79

4.1.2 Reviews ... 80

4.1.3 The General Process ... 82

4.1.4 Roles and Responsibilities ... 86

4.1.5 Types of Reviews ... 88

4.2 Static Analysis ... 95

4.2.1 The Compiler as a Static Analysis Tool ... 97

4.2.2 Examination of Compliance to Conventions and Standards ... 97

4.2.3 Execution of Data Flow Analysis ... 98

4.2.4 Execution of Control Flow Analysis ... 99

4.2.5 Determining Metrics ... 100

4.3 Summary ... 102

5 Dynamic Analysis – Test Design Techniques 105

5.1 Black Box Testing Techniques ... 110

5.1.1 Equivalence Class Partitioning ... 110

5.1.2 Boundary Value Analysis ... 121

5.1.3 State Transition Testing ... 128

5.1.4 Logic-Based Techniques (Cause-Effect Graphing and Decision Table Technique, Pairwise Testing) ... 136

5.1.5 Use-Case-Based Testing ... 141

5.1.6 General Discussion of the Black Box Technique ... 145

5.2 White Box Testing Techniques ... 145

5.2.1 Statement Testing and Coverage ... 146

5.2.2 Decision/Branch Testing and Coverage ... 148

5.2.3 Test of Conditions ... 151

5.2.4 Further White Box Techniques ... 159

5.2.5 General Discussion of the White Box Technique 160
5.2.6 Instrumentation and Tool Support 160
5.3 Intuitive and Experience-Based Test Case Determination 161
5.4 Summary 164

6 Test Management 169

6.1 Test Organization 169
6.1.1 Test Teams 169
6.1.2 Tasks and Qualifications 172
6.2 Planning 174
6.2.1 Quality Assurance Plan 174
6.2.2 Test Plan 175
6.2.3 Prioritizing Tests 177
6.2.4 Test Entry and Exit Criteria 179
6.3 Cost and Economy Aspects 180
6.3.1 Costs of Defects 180
6.3.2 Cost of Testing 181
6.3.3 Test Effort Estimation 184
6.4 Choosing the Test Strategy and Test Approach 184
6.4.1 Preventative vs. Reactive Approach 185
6.4.2 Analytical vs. Heuristic Approach 186
6.4.3 Testing and Risk 187
6.5 Managing The Test Work 189
6.5.1 Test Cycle Planning 189
6.5.2 Test Cycle Monitoring 190
6.5.3 Test Cycle Control 192
6.6 Incident Management 192
6.6.1 Test Log 193
6.6.2 Incident Reporting 193

6.6.3 Defect Classification 195
6.6.4 Incident Status 197
6.7 Requirements to Configuration Management 200
6.8 Relevant Standards 202
6.9 Summary 203

7 Test Tools 205

7.1 Types of Test Tools 205
7.1.1 Tools for Management and Control of Testing and Tests 206
7.1.2 Tools for Test Specification 209
7.1.3 Tools for Static Testing 210
7.1.4 Tools for Dynamic Testing 211
7.1.5 Tools for Nonfunctional Test 216
7.2 Selection and Introduction of Test Tools 218
7.2.1 Cost Effectiveness of Tool Introduction 219
7.2.2 Tool Selection 220
7.2.3 Tool Introduction 221
7.3 Summary 223

Appendix

A Test Plans According to IEEE Standard 829-1998 225
Test Plans According to IEEE Standard 829-2008 231

B Important Information about the Syllabus and the Certified Tester Exam 241

C Exercises 243

Glossary 247

Literature 277

Index 283

1 Introduction

In recent years, software been introduced virtually everywhere. There will soon be no appliances, machines, or facilities for which control is not implemented by software or software parts. In automobiles, for example, microprocessors and their accompanying software control more and more functionality, from engine management to the transmission and brakes. Thus, software is crucial to the correct functioning of devices and industry. Likewise, the smooth operation of an enterprise or organization depends largely on the reliability of the software systems used for supporting the business processes and particular tasks. How fast an insurance company can introduce a new product, or even a new rate, most likely depends on how quickly the IT systems can be adjusted or extended.

High dependence on the correct functioning of the software

Within both embedded and commercial software systems, quality has become the most important factor in determining success.

Many enterprises have recognized this dependence on software and strive for improved quality of their software systems and software engineering (or development) processes. One way to achieve this goal is through systematic evaluation and testing of the software. In some cases, appropriate testing procedures have found their way into the daily tasks associated with software development. However, in many sectors, there remains a significant need to learn about evaluation and testing.

Basic knowledge for structured evaluation and testing

With this book, we offer basic knowledge that will help you achieve structured and systematic evaluation and testing. Implementation of these evaluation and testing procedures should contribute to improvement of the quality of software. This book does not presume previous knowledge of software quality assurance. It is designed as a textbook and can even be used as a guide for self-study. We have included a single, continuous example to help provide an explanation and practical solutions for all of the topics we cover.

We want to help software testers who strive for a well-founded, basic knowledge of the principles behind software testing. We also address programmers and developers who are already performing testing tasks or will do so in the future. The book will help project managers and team leaders to improve the effectiveness and efficiency of software tests. Even those in disciplines related to IT, as well as employees who are involved in the processes of acceptance, introduction, and further development of IT applications, will find this book helpful for their daily tasks.

Evaluation and testing procedures are costly in practice (this area is estimated to consume 25% to 50% of software development time and cost [Koomen 99]). Yet, there are still too few universities, colleges, and vocational schools in the sectors of computer and information science that offer courses about this topic. This book will help both students and teachers. It provides the material for an introduction-level course.

Lifelong learning is indispensable, especially in the IT industry. Many companies and trainers offer further education in software testing to their employees. General recognition of a course certificate is possible, however, only if the contents of the course and the examination are defined and followed up by an independent body.

Certification program for software testers

In 1997, the Information Systems Examinations Board (ISEB) [URL: ISEB] of the British Computer Society (BCS) [URL: BCS] started a certification scheme to define course objectives for an examination (see the foreword by Dorothy Graham).

International initiative

Similar to the British example, other countries took up these activities and established independent, country-specific testing boards to make it possible to offer training and exams in the language of the respective countries. These national boards cooperate in the International Software Testing Qualifications Board (ISTQB) [URL: ISTQB]. An updated list of all ISTQB members can be found at [URL: ISTQB Members].

The ISTQB coordinates the national initiatives and assures uniformity and comparability of the courses and exam contents among the countries involved.

The national testing boards are responsible for issuing and maintaining curricula in the language of their countries and for organizing and executing examinations in their countries. They assess the seminars offered in their countries according to defined criteria and accredit training providers. The testing boards thus guarantee a high quality standard for the seminars. After passing an exam, the seminar participants receive an internationally recognized certificate of qualification.

Three-step qualification scheme

The ISTQB Certified Tester qualification scheme has three steps. The basics are described in the Foundation Level curriculum (syllabus). Building on this is the Advanced Level certificate, showing a deeper knowledge of testing and evaluation. The third level, the Expert Level, is intended for experienced professional software testers and consists of several modules about different special topics. Currently, the first four syllabi are being prepared in the ISTQB and the national boards. The syllabi for "Improving The Test Process" and "Test Management" are available. Syllabi for "Test Automation" and "Security Testing" are on their way. The current status of the syllabi can be seen at [URL: ISTQB].

The contents of this book correspond to the requirements of the ISTQB Foundation Level certificate. The knowledge needed to pass the exams can be acquired by self-study. The book can also be used to attain knowledge after, or parallel to, participation in a course.

The overall structure of this book corresponds to the course contents for the Foundation Level certificate.

Foundations

In chapter 2, "Fundamentals of Testing," the basics of software testing are discussed. In addition to the motivation for testing, the chapter will explain when to test, with which goals, and how intensively. The concept of a basic test process is described. The chapter shows the psychological difficulties experienced when testing one's own software and the problems that can occur when trying to find one's own errors.

Testing in the software life cycle

Chapter 3, "Testing in the Software Life Cycle," discusses which test activities should be performed during the software development process and when. In addition to describing the different test levels, it will examine the difference between functional and nonfunctional tests. Regression testing is also discussed.

Static testing

Chapter 4, "Static Test," discusses static testing techniques, that is, ways in which the test object is analyzed but not executed. Reviews and static analyses are already applied by many enterprises with positive results. This chapter will describe in detail the various methods and techniques.

Dynamic testing

Chapter 5, "Dynamic Analysis – Test Design Techniques," deals with testing in a narrower sense. The classification of dynamic testing techniques into black box and white box techniques will be discussed.

Each kind of test technique is explained in detail with the help of a continuous example. The end of the chapter shows the reasonable usage of exploratory and intuitive testing, which may be used in addition to the other techniques.

Test management

Chapter 6, “Test Management,” discusses aspects of test management such as systematic incident handling, configuration management, and testing economy.

Testing tools

Chapter 7, “Test Tools,” explains the different classes of tools that can be used to support testing. The chapter will include introductions to some of the tools and suggestions for selecting the right tools for your situation.

The appendices include additional information on the topics covered and for the exam.

Appendix A contains explanations of the test plan according to IEEE Standard 829-1998 [IEEE 829] and 829-2008. Appendix B includes important notes and additional information on the Certified Tester exam, and appendix C offers exercises to reinforce your understanding of the topics in each chapter. Finally, there is a glossary and a bibliography. Technical terms that appear in the glossary are marked with an arrow [➞] when they appear for the first time in the text. Text passages that go beyond the material of the syllabus are marked as “excursions.”

2 Fundamentals of Testing

This introductory chapter will explain basic facts of software testing, covering what you will need to know to understand the following chapters. Important concepts and essential vocabulary will be explained by using an example application that will be used throughout the book. It appears frequently to illustrate and clarify the subject matter. The fundamental test process with the different testing activities will be illustrated. Psychological problems with testing will be discussed. Finally, the ISTQB Code of Tester Ethics is presented and discussed.

Throughout this book, we'll use one example application to illustrate the software test methods and techniques presented in this book. The fundamental scenario is as follows.

Case study, "VirtualShowRoom" – VSR

A car manufacturer develops a new electronic sales support system called *VirtualShowRoom (VSR)*. The final version of this software system will be installed at every car dealer worldwide. Customers who are interested in purchasing a new car will be able to configure their favorite model (model, type, color, extras, etc.), with or without the guidance of a salesperson.

The system shows possible models and combinations of extra equipment and instantly calculates the price of the car the customer configures. A subsystem called *DreamCar* will provide this functionality.

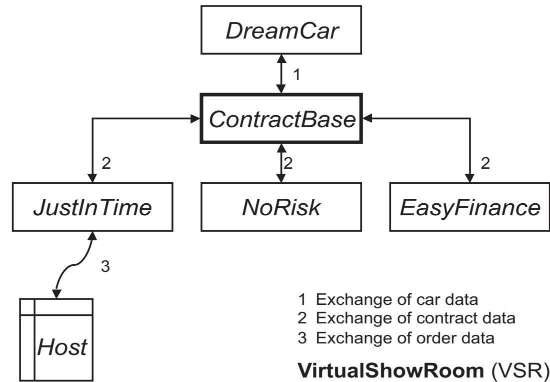
When the customer has made up her mind, she will be able to calculate the most suitable financing (*EasyFinance*) as well as place the order online (*JustInTime*). She will even get the option to sign up for the appropriate insurance (*NoRisk*). Personal information and contract data about the customer is managed by the *ContractBase* subsystem.

Figure 2-1 shows the general architecture of this software system.

Every subsystem will be designed and developed by a separate development team. Altogether, about 50 developers and additional employees from the respective user departments are involved in working on this project. External software companies will also participate.

The VSR-System must be tested thoroughly before release. The project members assigned to test the software apply different testing techniques and methods. This book contains the basic knowledge necessary for applying them.

Figure 2–1
Architecture of the VSR-System



2.1 Terms and Motivation

Requirements

During the construction of an industry product, the parts and the final product are usually examined to make sure they fulfill the given →requirements, that is, whether the product solves the required task.

Depending on the product, there may be different requirements to the →quality of the solution. If the product has problems, corrections must be made in the production process and/or in the design of the product itself.

Software is immaterial

What generally counts for the production of industry products is also appropriate for the production or development of software. However, testing (or evaluation) of partial products and the final product is more difficult, because a software product is not a tangible physical product. Direct examination is not possible. The only way to examine the product is by reading (reviewing) the development documents and code.

The dynamic behavior of the software, however, cannot be checked this way. It must be done through →testing, by executing the software on a computer. Its behavior must be compared to the requirements. Thus, testing of software is an important and difficult task in software development. It contributes to reducing the →risk of using the software because →defects can be found in testing. Testing and test documentation

are often defined in contracts, laws, or industrial or organizational standards.

Example

To identify and repair possible faults before delivery, the VSR-System from the case study example must be tested intensively before it is used. For example, if the system executes order transactions incorrectly, this could result in frustration for the customer and serious financial loss and a negative impact on the image of the dealer and the car manufacturer. Not finding such a defect constitutes a high risk during system use.

2.1.1 Error, Defect, and Bug Terminology

What is a defect, failure, or fault?

When does a system behave incorrectly, not conforming to requirements? A situation can be classified as incorrect only after we know what the correct situation is supposed to look like. Thus, a →failure means that a given requirement is not fulfilled; it is a discrepancy between the →actual result or behavior[1] and the →expected result or behavior.[2]

A failure is present if a legitimate (user) expectation is not adequately met. An example of a failure is a product that is too difficult to use or too slow but still fulfills the →functional requirements.

In contrast to physical system failure, software failures do not occur because of aging or abrasion. They occur because of →faults in the software. Faults (or defects or →bugs) in software are present from the time the software was developed or changed yet materialize only when the software is executed, becoming visible as a failure.

Failure

To describe the event when a user experiences a problem, [IEEE 610.12] uses the term *failure*. However, other terms, like *problem*, *issue*, and *incident*, are often used. During testing or use of the software, the failure becomes visible to the →tester or user; for example, an output is wrong or the program crashes.

Fault

We have to distinguish between the occurrence of a failure and its cause. A failure is caused by a fault in the software. This fault is also called a defect or internal error. Programmer slang for a fault is *bug*. For example, faults can be incorrect or forgotten →statements in the program.

Defect masking

It is possible that a fault is hidden by one or more other faults in other parts of the program (→defect masking). In that case, a failure occurs only

---

1. The actual behavior is identified while executing the test or during use of the system.
2. The expected behavior is defined in the specifications or requirements.

after the masking defects have been corrected. This demonstrates that corrections can have side effects.

One problem is that a fault can cause none, one, or many failures for any number of users and that the fault and the corresponding failure are arbitrarily far away from each other. A particularly dangerous example is some small corruption of stored data, which may be found a long time after it first occurred.

Error or mistake

The cause of a fault or defect is an →error or →mistake by a person—for example, defective programming by the developer. However, faults may even be caused by environmental conditions, like radiation and magnetism, that introduce hardware problems. Such problems are, however, not discussed in this book.

People err, especially under time pressure. Defects may occur, for example, by bad programming or incorrect use of program statements. Forgetting to implement a requirement leads to defective software. Another cause is changing a program part because it is complex and the programmer does not understand all consequences of the change. Infrastructure complexity, or the sheer number of system interactions, may be another cause. Using new technology often leads to defects in software, because the technology is not fully understood and thus not used correctly.

More detailed descriptions of the terms used in testing are given in the following section.

2.1.2 Testing Terms

Testing is not debugging

To be able to correct a defect or bug, it must be localized in the software. Initially, we know the effect of a defect but not the precise location in the software. Localization and correction of defects are tasks for a software developer and are often called →debugging. Repairing a defect generally increases the →quality of the product because the →change in most cases does not introduce new defects.

However, in practice, correcting defects often introduces one or more new defects. The new defects may then introduce failures for new, totally different inputs. Such unwanted side effects make testing more difficult. The result is that not only must we repeat the →test cases that have detected the defect, we must also conduct even more test cases to detect possible side effects.

Debugging is often equated with testing, but they are entirely different activities.

Debugging is the task of localizing and correcting faults. The goal of testing is the (more or less systematic) detection of failures (that indicate the presence of defects).

A test is a sample examination

Every execution[3] (even using more or less random samples) of a →test object in order to examine it is testing. The →test conditions must be defined. Comparing the actual and expected behaviors of the test object serves to determine if the test object fulfills the required characteristics.[4]

Testing software has different purposes:

- Executing a program to find failures
- Executing a program to measure quality
- Executing a program to provide confidence[5]
- Analyzing a program or its documentation to prevent failures

Tests can also be performed to acquire information about the test object, which is then used as the basis for decision-making—for example, about whether one part of a system is appropriate for integration with other parts of the system. The whole process of systematically executing programs to demonstrate the correct implementation of the requirements, to increase confidence, and to detect failures is called testing. In addition, a test includes static methods, that is, static analysis of software products using tools as well as document reviews (see chapter 4).

Testing terms

Besides execution of the test object with →test data, planning, design, implementation, and analysis of the test (→test management) also belong to the →test process. A →test run or →test suite includes execution of one or more →test cases. A test case contains defined test conditions. In most cases, these are the preconditions for execution, the inputs, and the expected outputs or the expected behavior of the test object. A test case should have a high probability of revealing previously unknown faults [Myers 79].

Several test cases can often be combined to create →test scenarios, whereby the result of one test case is used as the starting point for the next

---

3. This relates to dynamic testing (see chapter 5). In static testing (see chapter 4), the test object is not executed.
4. It is not possible to prove correct implementation of the requirements. We can only reduce the risk of serious bugs remaining in the program by testing.
5. If a thorough test finds few or no failures, confidence in the product will increase.

test case. For example, a test scenario for a database application can contain one test case writing a date into the database, another test case changing that date, and a third test case reading the changed date from the database and deleting it. (By deleting the date, the database should be in the same state as before executing this scenario.) Then all three test cases will be executed, one after another, all in a row.

No large software system is bug free

At present, there is no known bug-free software system, and there will probably not be any in the near future (if a system has nontrivial complexity). Often the reason for a fault is that certain exceptional cases were not considered during development and testing of the software. Such faults could be the incorrectly calculated leap year or the not-considered boundary condition for time behavior or needed resources. On the other hand, there are many software systems in many different fields that operate reliably, 24/7.

Testing cannot produce absence of defects

Even if all the executed test cases do not show any further failures, we cannot safely conclude (except for very small programs) that there are no further faults or that no further test cases could find them.

Excursion: Naming tests

There are many confusing terms for different kinds of software tests. Some will be explained later in connection with the description of the different →test levels (see chapter 3). The following terms describe the different ways tests are named:

→Test objective or test type:
A test is named according to its purpose (for example, →load test).

→Test technique:
A test is named according to the technique used for specifying or executing the test (for example, →business-process-based test).

Test object:
The name of a test reflects the kind of the test object to be tested (for example, a GUI test or DB test [database test]).

Test level:
A test is named after the level of the underlying life cycle model (for example, →system test).

Test person:
A test is named after the personnel group executing the tests (for example, developer test, →user acceptance test).

Test extent:
A test is named after the level of extent (for example, partial →regression test, full test).

Thus, not every term means a new or different kind of testing. In fact, only one of the aspects is pushed to the fore. It depends on the perspective we use when we look at the actual test.

2.1.3 Software Quality

Software testing contributes to improvement of →software quality. This is done by identifying defects and subsequently correcting them. If the test cases are a reasonable sample of software use, quality experienced by the user should not be too different from quality experienced during testing.

But software quality is more than just the elimination of failures found during testing. According to the ISO/IEC Standard 9126-1 [ISO 9126], software quality comprises the following factors:

→functionality, →reliability, usability, →efficiency, →maintainability, and portability.

Testing must consider all these factors, also called →quality characteristics and →quality attributes, in order to judge the overall quality of a software product. Which quality level the test object is supposed to show for each characteristic should be defined in advance. Appropriate tests must then check to make sure these requirements are fulfilled.

Excursion: ISO/IEC 25010

In 2011 ISO/IEC Standard 9126 was replaced by ISO/IEC Standard 25010 [ISO 25010]. The current ISTQB syllabus still refers to ISO/IEC 9126. Here is a short overview of the new standard.

ISO/IEC 25010 partitions software quality into three models: quality in use model, product quality model, and data quality model. The quality in use model comprises the following characteristics: effectiveness, satisfaction, freedom from risk, and context coverage. The product quality model comprises functional sustainability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. In this area much is like in ISO/IEC 9126. Data quality is defined in ISO/IEC 25012 [ISO 25012].

Example VirtualShowRoom

In the case of the VSR-System, the customer must define which of the quality characteristics are important. Those must be implemented in the system and then checked for. The characteristics of functionality, reliability, and usability are very important for the car manufacturer. The system must reliably provide the required functionality. Beyond that, it must be easy to use so that the different car dealers can use it without any problems in everyday life. These quality characteristics should be especially well tested in the product.

We discuss the individual quality characteristics of ISO/IEC Standard 9126-1 [ISO 9126] in the following section.

Functionality

When we talk about functionality, we are referring to all of the required capabilities of a system. The capabilities are usually described by a specific input/output behavior and/or an appropriate reaction to an

input. The goal of the test is to prove that every single required capability in the system was implemented as described in the specifications. According to ISO/IEC Standard 9126-1, the functionality characteristic contains the subcharacteristics adequacy, accuracy, interoperability, correctness, and security.

An appropriate solution is achieved if every required capability is implemented in the system. Thereby it is clearly important to pay attention to, and thus to examine during testing, whether the system delivers the correct or specified outputs or effects.

Software systems must interoperate with other systems, at least with the operating system (unless the operating system is the test object itself).

Interoperability describes the cooperation between the system to be tested and other specified systems. Testing should detect trouble with this cooperation.

Adequate functionality also requires fulfilling usage-specific standards, contracts, rules, laws, and so on. Security aspects such as access control and →data security are important for many applications. Testing must show that intentional and unintentional unauthorized access to programs and data is prevented.

Reliability

Reliability describes the ability of a system to keep functioning under specific use over a specific period. In the standard, the reliability characteristic is split into maturity, →fault tolerance, and recoverability.

Maturity means how often a failure of the software occurs as a result of defects in the software.

Fault tolerance is the capability of the software product to maintain a specified level of performance or to recover from faults such as software faults, environment failures, wrong use of interface, or incorrect input.

Recoverability is the capability of the software product to reestablish a specified level of performance (fast and easily) and recover the data directly affected in case of failure. Recoverability describes the length of time it takes to recover, the ease of recovery, and the amount of work required to recover. All this should be part of the test.

Usability

Usability is very important for acceptance of interactive software systems. Users won't accept a system that is hard to use. What is the effort required for the usage of the software for different user groups? Understandability, ease of learning, operability, and attractiveness as well as compliance to standards, conventions, style guides, and user interface regulations are aspects of usability. These quality characteristics are checked in →nonfunctional tests (see chapter 3).

Efficiency

Efficiency tests may give measurable results. An efficiency test measures the required time and consumption of resources for the execution of tasks. Resources may include other software products, the software and hardware →configuration of the system, and materials (for example, print paper, network, and storage).

Maintainability and portability

Software systems are often used over a long period on various platforms (operating system and hardware). Therefore, the last two quality criteria are very important: maintainability and portability.

Subcharacteristics of maintainability are analyzability, changeability, stability, and testability.

Subcharacteristics of portability are adaptability, ease of installation, conformity, and interchangeability. Many aspects of maintainability and portability can only be examined by →static analysis (see section 4.2).

A software system cannot fulfill every quality characteristic equally well. Sometimes it is possible that meeting one characteristic results in a conflict with another one. For example, a highly efficient software system can become hard to port because the developers usually use special characteristics (or features) of the chosen platform to improve efficiency. This in turn negatively affects portability.

Prioritize quality characteristics

Quality characteristics must therefore be prioritized. The quality specification is used to determine the test intensity for the different quality characteristics. The next chapter will discuss the amount of work involved in these tests.

2.1.4 Test Effort

Complete testing is impossible

Testing cannot prove the absence of faults. In order to do this, a test would need to execute a program in every possible situation with every possible input value and with all possible conditions. In practice, a →complete or exhaustive test is not feasible. Due to combinational effects, the outcome of this is an almost infinite number of tests. Such a “testing” for all combinations is not possible.

Example

The fact that complete testing is impossible is illustrated by an example of →control flow testing [Myers 79].

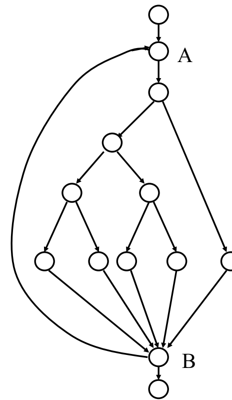
A small program with an easy control flow will be tested. The program consists of four decisions (IF-instructions) that are partially nested. The control flow graph of the program is shown in figure 2-2. Between Point A and B is a loop, with a return from Point B to Point A. If the program is supposed to be exhaustively tested for the different control-flow-based possibilities, every possible

flow—i.e., every possible combination of program parts—must be executed. At a loop limit of a maximum of 20 cycles and considering that all links are independent, the outcome is the following calculation, whereby 5 is the number of possible ways within the loop:

$$5^{20} + 5^{19} + 5^{18} + \ldots + 5^{1}$$

5^1 test cases result from execution of every single possible way within the loop, but in each case without return to the loop starting point. If the test cases result in one single return to the loop starting point, then $5 \times 5 = 5^2$ different possibilities must be considered, and so on. The total result of this calculation is about 100 quadrillion different sequences of the program.

Figure 2–2
Control flow graph of a small program



Assuming that the test is done manually and a test case, as Myers describes [Myers 79], takes five minutes to specify, to execute, and to be analyzed, the time for this test would be one billion years. If we assume five microseconds instead of five minutes per test case, because the test mainly runs automatically, it would still last 19 years.

Test effort between 25% and 50%

Thus, in practice it is not possible to test even a small program exhaustively.

It is only possible to consider a part of all imaginable test cases. But even so, testing still accounts for a large portion of the development effort. However, a generalization of the extent of the →test effort is difficult because it depends very much on the character of the project. The following list shows some example data from projects of one large German software company. This should shed light on the spectrum of different testing efforts relative to the total budget of the development.

- For some major projects with more than 10 person-years' effort, coding and testing together used 40%, and a further 8% was used for the integration. At test-intensive projects (for example, →safety-critical systems), the testing effort increased to as much as 80% of the total budget.
- In one project, the testing effort was 1.2 times as high as the coding effort, with two-thirds of the test effort used for →component testing.
- For another project at the same software development company, the system test cost was 51.9% of the project.

Test effort is often shown as the proportion between the number of testers and the number of developers. The proportion varies from 1 tester per 10 developers to up to 3 testers per developer. The conclusion is that test efforts or the budget spent for testing vary enormously.

Defects can cause high costs

But is this high testing effort affordable and justifiable? The counter question from Jerry Weinberg is "Compared to what?" [DeMarco 93]. His question refers to the risks of faulty software systems. Risk is calculated as the probability of occurrence and the expected amount of damage.

Faults that were not found during testing can cause high costs when the software is used. The German newspaper *Frankfurter Allgemeine Zeitung* from January 17, 2002, had an article titled "IT system breakdowns cost many millions." A one-hour system breakdown in the stock exchange is estimated to cost $7.8 million. When safety-critical systems fail, the lives and health of people may be in danger.

Since a full test is not possible, the testing effort must have an appropriate relation to the attainable result. "Testing should continue as long as costs of finding and correcting a defect[6] are lower than the costs of failure" [Koomen 99]. Thus, the test effort is always dependent on an estimation of the application risk.

Example for a high risk in case of failure

In the case of the VSR-System, the prospective customers configure their favorite car model on the display. If the system calculates a wrong price, the customer can insist on that price. In a later stage of the VSR-System, the company plans to offer a web-based sales portal. In that case, a wrong price can lead to thousands of cars being sold for a price that's too low. The total loss can amount to millions, depending on how much the price was miscalculated by the VSR-System. The legal view is that an online order is a valid sales contract with the quoted price.

6. The cost must include all aspects of a failure, even the possible cost of bad publicity, litigation, etc., and not just the cost of correction, retesting, and distribution.

Systems with high risks must be tested more thoroughly than systems that do not generate big losses if they fail. The risk assessment must be done for the individual system parts, or even for single error possibilities. If there is a high risk for failures by a system or subsystem, there must be a greater testing effort than for less critical (sub)systems. International standards for production of safety-critical systems use this approach to require that different test techniques be applied for software of different integrity levels.

For a producer of a computer game, saving erroneous game scores can mean a very high risk, even if no real damage is done, because the customers will not trust a defective game. This leads to high losses of sales, maybe even for all games produced by the company.

Define test intensity and test extent depending on risk

Thus, for every software program it must be decided how intensively and thoroughly it shall be tested. This decision must be made based upon the expected risk of failure of the program. Since a complete test is not possible, it is important how the limited test resources are used. To get a satisfying result, the tests must be designed and executed in a structured and systematic way. Only then is it possible to find many failures with appropriate effort and avoid →unnecessary tests that would not give more information about system quality.

Select adequate test techniques

There exist many different methods and techniques for testing software.

Every technique especially focuses on and checks particular aspects of the test object. Thus, the focus of examination for the control-flow-based test techniques is the program flow. In case of the →data flow test techniques, the examination focuses on the use and flow of data. Every test technique has its strengths and weaknesses in finding different kinds of faults. There is no test technique that is equally well suited for all aspects. Therefore, a combination of different test techniques is always necessary to detect failures with different causes.

Test of extra functionality

During the test execution phase, the test object is checked to determine if it works as required by the →specifications. It is also important—and thus naturally examined while testing—that the test object does not execute functions that go beyond the requirements. The product should provide only the required functionality.

Test case explosion

The testing effort can grow very large. Test managers face the dilemma of possible test cases and test case variants quickly becoming hundreds or thousands of tests. This problem is also called combinatorial explosion, or →test case explosion. Besides the necessary restriction in the number of

test cases, the test manager normally has to fight with another problem: lack of resources.

Limited resources

Participants in every software development project will sooner or later experience a fight about resources. The complexity of the development task is underestimated, the development team is delayed, the customer pushes for an earlier release, or the project leader wants to deliver "something" as soon as possible. The test manager usually has the worst position in this "game." Often there is only a small time window just before delivery for executing the test cases and very few testers are available to run the test. It is certain that the test manager does not have the time and resources for executing an "astronomical" amount of test cases.

However, it is expected that the test manager delivers trustworthy results and makes sure the software is sufficiently tested. Only if the test manager has a well-planned, efficient strategy is there is a chance to fulfill this challenge successfully. A fundamental test process is required. Besides the adherence to a fundamental test process, further →quality assurance activities must be accomplished, such as, for example, →reviews (see section 4.1.2). Additionally, a test manager should learn from earlier projects and improve the development and testing process.

The next section describes a fundamental test process typically used for the development and testing of systems like the VSR-System.

2.2 The Fundamental Test Process

Excursion
Life cycle models

To accomplish a structured and controllable software development effort, software development models and →development processes are used. Many different models exist. Examples are the waterfall model [Boehm 73], [Boehm 81], the general V-model[7] [Boehm 79], and the German V-model XT [URL: V-model XT]). Furthermore, there are the spiral model, different incremental or evolutionary models, and the agile, or lightweight, methods like XP (Extreme Programming [Beck 00]) and SCRUM [Beedle 01], which are popular nowadays (for example, see [Bleek 08]). Development of object-oriented software systems often uses the rational unified process [Jacobson 99].

All of these models define a systematic, orderly way of working during the project. In most cases, phases or design steps are defined. They have to be completed with a result in the form of a document. A phase completion, often called a →milestone, is achieved when the required documents are completed and conform to the given quality criteria. Usually, →roles dedicated to specific tasks in software development

7. The general V-model will be referred to as the general model to make sure it is not confused with the German V-model, referred to as just V-model.

are defined. Project staff has to accomplish these tasks. Sometimes, the models even define the techniques and processes to be used in a particular phase. With the aid of these models, detailed planning of resource usage (time, personnel, infrastructure, etc.) can be performed. In a project, the development models define the collective and mandatory tasks and their chronological sequence.

Testing appears in each of these life cycle models, but with very different meanings and to a different extent. In the following, some models will be briefly discussed from the view of testing.

The waterfall model: Testing as "final inspection"

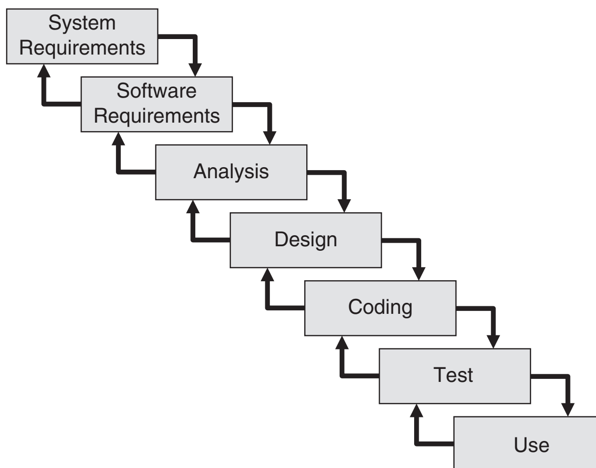
The first fundamental model was the waterfall model (see figure 2-3, shown with the originally defined phases [Royce 70][8]). It is impressively simple and very well known. Only when one development phase is completed will the next one be initiated.

Between adjacent phases only, there are feedback loops that allow, if necessary, required revisions in the previous phase. The crucial disadvantage of this model is that testing is understood as a "one time" action at the end of the project just before the release to operation. The test is seen as a "final inspection," an analogy to a manufacturing inspection before handing over the product to the customer.

The general V-model

An enhancement of the waterfall model is the general V-model ([Boehm 79], [IEEE/IEC 12207]), where the constructive activities are decomposed from the testing activities (see chapter 3, figure 3-1). The model has the form of a V. The constructive activities, from requirements definition to implementation, are found on the downward branch of the V. The test execution activities on the ascending branch are organized by test levels and matched to the appropriate abstraction level on the opposite side's constructive activity. The general V-model is common and frequently used in practice.

Figure 2–3
Waterfall-model



8. Royce did not call his model a waterfall model. He said in his paper, "Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps."

The description of tasks in the process models discussed previously is not sufficient as an instruction on how to perform structured tests in software projects. In addition to embedding testing in the whole development process, a more detailed process for the testing tasks themselves is needed (see figure 2-4). This means that the "content" of the development task testing must be split into smaller subtasks, as follows: →test planning and control, test analysis and design, test implementation and execution, evaluation of test →exit criteria and reporting, and test closure activities. Although illustrated sequentially, the activities in the test process may overlap or take place concurrently. Test activities also need to be adjusted to the individual needs of each project. The test process described here is a generic one. The listed subtasks form a fundamental test process and are described in more detail in the following sections.

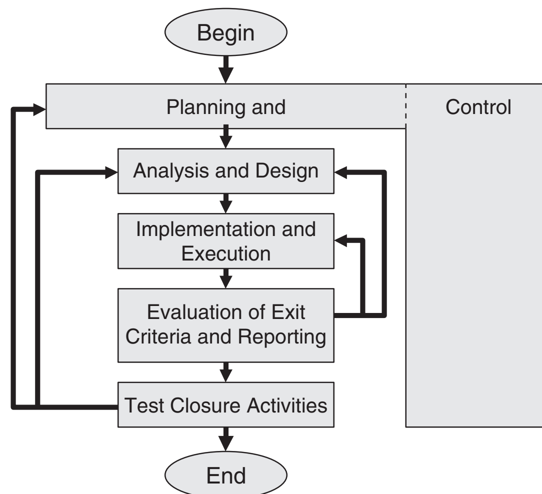


Figure 2–4
ISTQB fundamental test process

2.2.1 Test Planning and Control

Execution of such a substantial task as testing must not take place without a plan. Planning of the test process starts at the beginning of the software development project. As with all planning, during the course of the project the previous plans must be regularly checked, updated, and adjusted.

Resource planning

The mission and objectives of testing must be defined and agreed upon as well as the resources necessary for the test process. Which employees are needed for the execution of which tasks and when? How much time is needed, and which equipment and utilities must be availa-

ble? These questions and many more must be answered during planning, and the result should be documented in the →test plan (see chapter 6). Necessary training programs for the employees should be prepared. An organizational structure with the appropriate test management must be arranged or adjusted if necessary.

Test control is the monitoring of the test activities and comparing what actually happens during the project with the plan. It includes reporting the status of deviations from the plan and taking any actions necessary to meet the planned goals in the new situation. The test plan must be updated to the changed situation.

Part of the test management tasks is administrating and maintaining the test process, the →test infrastructure, and the →testware. Progress tracking can be based on appropriate reporting from the employees as well as data automatically generated from tools. Agreements about these topics must be made early.

Determination of the test strategy

The main task of planning is to determine the →test strategy or approach (see section 6.4). Since an exhaustive test is not possible, priorities must be set based on risk assessment. The test activities must be distributed to the individual subsystems, depending on the expected risk and the severity of failure effects. Critical subsystems must get greater attention, thus be tested more intensively. For less critical subsystems, less extensive testing may be sufficient. If no negative effects are expected in the event of a failure, testing could even be skipped on some parts. However, this decision must be made with great care. The goal of the test strategy is the optimal distribution of the tests to the "right" parts of the software system.

Example for a test strategy

The VSR-System consists of the following subsystems:

- *DreamCar* allows the individual configuration of a car and its extra equipment.
- *ContractBase* manages all customer information and contract data.
- *JustInTime* implements the ability to place online orders (within the first expansion stage by the dealer).
- *EasyFinance* calculates an optimal method of financing for the customer.
- *NoRisk* provides the ability to purchase appropriate insurance.

Naturally, the five subsystems should not be tested with identical intensity. The result of a discussion with the VSR-System client is that incorrect behavior of the *DreamCar* and *ContractBase* subsystems will have the most harmful effects. Because of this, the test strategy dictates that these two subsystems must be tested more intensively.

The possibility to place orders online, provided by the subsystem *JustInTime*, is found to be less critical because the order can, in the worst case, still be passed on in other ways (via fax, for example). But it is important that the order data must not be altered or get lost in the *JustInTime* subsystem. Thus, this aspect should be tested more intensively.

For the other two subsystems, *NoRisk* and *EasyFinance*, the test strategy defines that all of their main functions (computing a rate, recording and placing contracts, saving and printing contracts, etc.) must be tested. Because of time constraints, it is not possible to cover all conceivable contract variants for financing and insuring a car. Thus, it is decided to concentrate the test around the most commonly occurring rate combinations. Combinations that occur less frequently get a lower priority (see sections 6.2 and 6.4).

With these first thoughts about the test strategy for the VSR-System, it is clear that it is reasonable to choose the level of intensity for testing whole subsystems as well as single aspects of a system.

Define test intensity for subsystems and different aspects

The intensity of testing depends very much on the test techniques that are used and the →test coverage that must be achieved. Test coverage serves as a test exit criterion. Besides →coverage criteria referring to source code structure (for example, statement coverage; see section 5.2), it is possible to define meeting the customer requirements as an exit criterion. It may be demanded that all functions must be tested at least once or, for example, that at least 70% of the possible transactions in a system are executed. Of course, the risk in case of failure should be considered when the exit criteria, and thus the intensity of the tests, are defined. Once all test exit criteria[9] are defined, they may be used after executing the test cases to decide if the test process can be finished.

Prioritization of the tests

Because software projects are often run under severe time pressure, it is reasonable to appropriately consider the time aspect during planning. The prioritization of tests guarantees that the critical software parts are tested first in case time constraints do not allow executing all the planned tests (see section 6.2).

Tool support

If the necessary tool support (see chapter 7) does not exist, selection and acquisition of tools must be initiated early. Existing tools must be evaluated if they are updated. If parts of the test infrastructure have to be developed, this can be prepared. →Test harnesses (or →test beds), where subsystems can be executed in isolation, must often be programmed. They must be created soon enough to be ready after the respective test objects

9. Another term is *test end criteria*.

are programmed. If frameworks—such as Junit [URL: xunit]—shall be applied, their usage should be announced early in the project and should be tried in advance.

2.2.2 Test Analysis and Design

Review the test basis

The first task is to review the →test basis, i.e., the specification of what should be tested. The specification should be concrete and clear enough to develop test cases. The basis for the creation of a test can be the specification or architecture documents, the results of risk analysis, or other documents produced during the software development process.

For example, a requirement may be too imprecise in defining the expected output or the expected behavior of the system. No test cases can then be developed. →Testability of this requirement is insufficient. Therefore it must be reworked. Determining the →preconditions and requirements to test case design should be based on an analysis of the requirements, the expected behavior, and the structure of the test object.

Check testability

As with analyzing the basis for a test, the test object itself also has to fulfill certain requirements to be simple to test. Testability has to be checked. This process includes checking the ease with which interfaces can be addressed (interface openness) and the ease with which the test object can be separated into smaller, more easily testable units. These issues need to be addressed during development and the test object should be designed and programmed accordingly. The results of this analysis are also used to state and prioritize the test conditions based on the general objectives of the test. The test conditions state exactly what shall be tested. This may be a function, a component, or some quality characteristic.

Consider the risk

The test strategy determined in the test plan defines which test techniques shall be used. The test strategy is dependent on requirements for reliability and safety. If there is a high risk of failure for the software, very thorough testing should be planned. If the software is less critical, testing may be less formal.

In the →test specification, the test cases are then developed using the test techniques specified. Techniques planned previously are used, as well as techniques chosen based on an analysis of possible complexity in the test object.

Traceability is important

It is important to ensure →traceability between the specifications to be tested and the tests themselves. It must be clear which test cases test which requirements and vice versa. Only this way is it possible to decide which requirements are to be or have been tested, how intensively and

with which test cases. Even the traceability of requirement changes to the test cases and vice versa should be verified.

Logical and concrete test cases

Specification of the test cases takes place in two steps. →Logical test cases have to be defined first. After that, the logical test cases can be translated into concrete, physical test cases, meaning the actual inputs are selected (→concrete test cases). Also, the opposite sequence is possible: from concrete to the general logical test cases. This procedure must be used if a test object is specified insufficiently and test specification must be done in a rather experimental way (→exploratory testing, see section 5.3). Development of physical test cases, however, is part of the next phase, test implementation.

The test basis guides the selection of logical test cases with all test techniques. The test cases can be determined from the test object's specification (→black box test design techniques) or be created by analyzing the source code (→white box test design techniques). It becomes clear that the activity called →test case specification can take place at totally different times during the software development process. This depends on the chosen test techniques, which are found in the test strategy. The process models shown at the beginning of section 2.2 represent the test execution phases only. Test planning, analysis, and design tasks can and should take place in parallel with earlier development activities.

Test cases comprise more than just the test data

For each test case, the initial situation (precondition) must be described. It must be clear which environmental conditions must be fulfilled for the test. Furthermore, before →test execution, it must be defined which results and behaviors are expected. The results include outputs, changes to global (persistent) data and states, and any other consequences of the test case.

Test oracle

To define the expected results, the tester must obtain the information from some adequate source. In this context, this is often called an oracle, or →test oracle. A test oracle is a mechanism for predicting the expected results. The specification can serve as a test oracle. There are two main possibilities:

- The tester derives the expected data based on the specification of the test object.
- If functions doing the reverse action are available, they can be run after the test and then the result is verified against the original input. An example of this scenario is encryption and decryption of data.

See also chapter 5 for more information about predicting the expected results.

Test cases for expected and unexpected inputs

Test cases can be differentiated by two criteria:

- First are test cases for examining the specified behavior, output, and reaction. Included here are test cases that examine specified handling of exception and error cases (→negative test). But it is often difficult to create the necessary preconditions for the execution of these test cases (for example, capacity overload of a network connection).
- Next are test cases for examining the reaction of test objects to invalid and unexpected inputs or conditions, which have no specified →exception handling.

Example for test cases

The following example is intended to clarify the difference between logical and concrete (physical) test cases.

Using the sales software, the car dealer is able to define discount rules for his salespeople: With a price of less than $15.000, no discount shall be given. For a price of $20.000, 5% is OK. If the price is below $25.000, a 7% discount is possible. For higher prices, 8.5% can be granted.

From this, the following cases can be derived:

Price < 15.000 discount = 0%
15.000 ≤ price ≤ 20.000 discount = 5%
20.000 < price < 25.000 discount = 7%
price ≥ 25.000 discount = 8.5%

It becomes obvious that the text has room for interpretation[10], which may be misunderstood. With more formal, mathematical description, this will not happen. However, the discounts are clearly stated. From the more formal statement (above), table 2-1 can be developed.

Table 2–1
Table with logical test cases

Logical test case	1	2	3	4
input value x (price in dollar)	x < 15000	15000 ≤ x ≤ 20000	20000 < x < 25000	x ≥ 25000
predicted result (discount in %)	0	5	7	8.5

10. In the preceding paragraph, it is unclear what happens at exactly 25.000.

To execute the test cases, the logical test cases must be converted into concrete test cases. Concrete inputs must be chosen (see table 2-2) Special preconditions or conditions are not given for these test cases.

Table 2–2
Table with concrete test cases

Concrete test case	1	2	3	4
input value x (price in dollar)	14500	16500	24750	31800
predicted result (discount in %)	0	825	1732.50	2703

The values chosen here shall only serve to illustrate the difference between logical and concrete test cases. No explicit test method has been used for designing them. We do not claim that the program is tested well enough with these four test cases. For example, there are no test cases for wrong inputs, such as, for example, negative prices. More detailed descriptions of methods for designing test cases are given in chapter 5.

In parallel to the described test case specification, it is important to decide on and prepare the test infrastructure and the necessary environment to execute the test object. To prevent delays during test execution, the test infrastructure should already be assembled, integrated, and verified as much as possible at this time.

2.2.3 Test Implementation and Execution

Here, logical test cases must be transformed into concrete test cases; all the details of the environment (test infrastructure and test framework) must be set up. The tests must be run and logged.

When the test process has advanced and there is more clarity about technical implementation, the logical test cases are converted into concrete ones. These test cases can then be used without further modifications or additions for executing the test, if the defined →preconditions for the respective test case are fulfilled. The mutual traceability between test cases and specifications must be checked and, if necessary, updated.

Test case execution

In addition to defining test cases, one must describe how the tests will be executed. The priority of the test cases (see section 6.2.3), decided during test planning, must be taken into account. If the test developer executes the tests himself, additional, detailed descriptions may not be necessary.

The test cases should also be grouped into →test suites or test scenarios for efficient test execution and easier understanding.

Test harness

In many cases specific test harnesses, →drivers, →simulators, etc. must be programmed, built, acquired, or set up as part of the test environment before the test cases can be executed. Because failures may also be caused by faults in the test harness, the →test environment must be checked to make sure it's working correctly.

When all preparatory tasks for the test have been accomplished, test execution can start immediately after programming and delivery of the subsystems to testing. Test execution may be done manually or with tools using the prepared sequences and scenarios.

Checking for completeness

First, the parts to be tested are checked for completeness. The test object is installed in the available test environment and tested for its ability to start and do the main processing.

Examination of the main functions

The recommendation is to start test execution with the examination of the test object's main functionality (→smoke test). If →failures or →deviations from the expected result show up at this time, it is foolish to continue testing. The failures or deviations should be corrected first. After the test object passes this test, everything else is tested. Such a sequence should be defined in the test approach.

Tests without a log are of no value

Test execution must be exactly and completely logged. This includes logging which test runs have been executed with which results (pass or failure). On the one hand, the testing done must be comprehensible to people not directly involved (for example, the customer) on the basis of these →test logs. On the other hand, the execution of the planned tests must be provable. The test log must document who tested which parts, when, how intensively, and with what results.

Reproducibility is important

Besides the test object, quite a number of documents and pieces of information belong to each test execution: test environment, input data, test logs, etc. The information related to a test case or test run must be maintained in such a way that it is possible to easily repeat the test later with the same input data and conditions. The testware must be subjected to →configuration management (see also section 6.7).

Failure found?

If a difference shows up between expected and actual results during test execution, it must be decided when evaluating the test logs if the difference really indicates a failure. If so, the failure must be documented. At first, a rough analysis of possible causes must be made. This analysis may require the tester to specify and execute additional test cases.

The cause for a failure can also be an erroneous or inexact test specification, problems with the test infrastructure or the test case, or an incorrect test execution. The tester must examine carefully if any of these pos-

sibilities apply. Nothing is more detrimental to the credibility of a tester than reporting a supposed failure whose cause is actually a test problem. But the fear of this possibility should not result in potential failures not being reported, i.e., the testers starting to self-censor their results. This could be fatal as well.

In addition to reporting discrepancies between expected and real results, test coverage should be measured (see section 2.2.4). If necessary, the use of time should also be logged. The appropriate tools for this purpose should be used (see chapter 7).

Correction may lead to new faults

Based on the →severity of a failure (see section 6.6.3), a decision must be made about how to prioritize fault corrections. After faults are corrected, the tester must make sure the fault has really been corrected and that no new faults have been introduced (see section 3.7.4). New testing activities result from the action taken for each incident—for example, re-execution of a test that previously failed in order to confirm a defect fix, execution of a corrected test, and/or regression tests. If necessary, new test cases must be specified to examine the modified or new source code. It would be convenient to correct faults and retest corrections individually to avoid unwanted interactions of the changes. In practice, this is not often possible. If the test is not executed by the developer, but instead by independent testers, a separate correction of individual faults is not practical or possible. It would take a prohibitive amount of effort to report every failure in isolation to the developer and continue testing only after corrections are made. In this case, several defects are corrected together and then a new software version is installed for new testing.

The most important test cases first

In many projects, there is not enough time to execute all specified test cases. When that happens, a reasonable selection of test cases must be made to make sure that as many critical failures as possible are detected. Therefore, test cases should be prioritized. If the tests end prematurely, the best possible result should be achieved. This is called →risk-based testing (see section 6.4.3).

Furthermore, an advantage of assigning priority is that important test cases are executed first, and thus important problems are found and corrected early. An equal distribution of the limited test resources on all test objects of the project is not reasonable. Critical and uncritical program parts are then tested with the same intensity. Critical parts would be tested insufficiently, and resources would be wasted on uncritical parts for no reason.

2.2.4 Test Evaluation and Reporting[11]

End of test?

During test evaluation and reporting, the test object is assessed against the set test exit criteria specified during planning. This may result in normal termination of the tests if all criteria are met, or it may be decided that additional test cases should be run or that the criteria were too hard.

It must be decided whether the test exit criteria defined in the test plan are fulfilled.

Considering the risk, an adequate exit criterion must be determined for each test technique used. For example, it could be specified that a test is considered good enough after execution of 80% of the test object statements. However, this would not be a very high requirement for a test. Appropriate tools should be used to collect such measures, or →metrics, in order to decide when a test should end (see section 7.1.4).

If at least one test exit criterion is not fulfilled after all tests are executed, further tests must be executed. Attention should be paid to ensure that the new test cases better cover the respective exit criteria. Otherwise, the extra test cases just result in additional work but no improvement concerning the end of testing.

Is further effort justifiable?

A closer analysis of the problem can also show that the necessary effort to fulfill the exit criteria is not appropriate. In that situation, further tests are canceled. Such a decision must, naturally, consider the associated risk.

An example of such a case may be the treatment of an exceptional situation. With the available test environment, it may not be possible to introduce or simulate this situation. The appropriate source code for treating it can then not be executed and tested. In such cases, other examination techniques should be used, such as, for example, static analysis (see section 4.2).

Dead code

A further case of not meeting test exit criteria may occur if the specified criterion is impossible to fulfill in the specific case. If, for example, the test object contains →dead code, then this code cannot be executed. Thus, 100% statement coverage is not possible because this would also include the unreachable (dead) code. This possibility must be considered in order to avoid further senseless tests trying to fulfill the criterion. An impossible criterion is often a hint to possible inconsistent or imprecise requirements or specifications. For example, it would certainly make sense to investigate

11. ISTQB calls this phase "Evaluation of test exit criteria and Reporting."

why the program contains instructions that cannot be executed. Doing this allows further faults to be found so their corresponding failures can be prevented.

If further tests are planned, the test process must be resumed, and it must be decided at which point the test process will be reentered. Sometimes it is even necessary to revise the test plan because additional resources are needed. It is also possible that the test specifications must be improved in order to fulfill the required exit criterion.

Further criteria for the determination of the test's end

In addition to test coverage criteria, other criteria can be used to define the test's end. A possible criterion is the failure rate. Figure 2-5 shows the average number of new failures per testing hour over 10 weeks. In the 1st week, there was an average of two new failures per testing hour. In the 10th week, it is fewer than one failure per two hours. If the failure rate falls below a given threshold (e.g., fewer than one failure per testing hour), it will be assumed that more testing is not economically justified and the test can be ended.

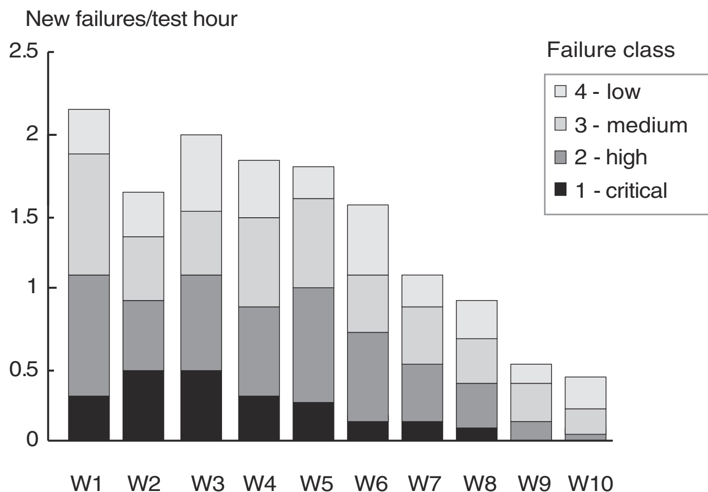


Figure 2–5
Failure rate

When deciding to stop testing this way it must be considered that some failures can have very different effects. Classifying and differentiating failures according to their impact to the stakeholders (i.e., failure severity) is therefore reasonable and should generally be considered (see section 6.6.3).

Consider several test cycles

The failures found during the test should be repaired, after which a new test becomes necessary. If further failures occur during the new test,

new test cycles may be necessary. Not planning such correction and testing cycles by assuming that no failures will occur while testing is unrealistic. Because it can be assumed that testing finds failures, additional faults must be removed and retested in a further →test cycle. If this cycle is ignored, then the project will be delayed. The required effort for defect correction and the following cycles is difficult to calculate. Historical data from previous, similar projects can help. The project plan should provide for the appropriate time buffers and personnel resources.

End criteria in practice: Time and cost

In practice, the end of a test is often defined by factors that have no direct connection to the test: time and costs. If these factors lead to stopping the test activities, it is because not enough resources were provided in the project plan or the effort for an adequate test was underestimated.

Successful testing saves costs

Even if testing consumed more resources than planned, it nevertheless results in savings due to elimination of faults in the software. Faults delivered in the product mostly cause higher costs when found during operation (see section 6.3.1).

Test summary report

When the test criteria are fulfilled or a deviation from them is clarified, a →test summary report should be written for the stakeholders, which may include the project manager, the test manager, and possibly the customer. In lower-level tests (component tests), this may just take the form of a message to the project manager about meeting the criteria. In higher-level tests, a formal report may be required.

2.2.5 Test Closure Activities

Learning from experience

It is a pity that these activities, which should be executed during this final phase in the test process, are often left out. The experience gathered during the test work should be analyzed and made available for future projects. Of interest are deviations between planning and execution for the different activities as well as the assumed causes. For example, the following data should be recorded:

- When was the software system released?
- When was the test finished or terminated?
- When was a milestone reached or a maintenance release completed?

Important information for evaluation can be extracted by asking the following questions:

- Which planned results were achieved and when—if at all?
- Which unexpected events happened (reasons and how they were met)?

- Are there any open problems and →change requests? Why were they not implemented?
- How was user acceptance after deploying the system?

The evaluation of the test process—i.e., a critical evaluation of the executed tasks in the test process, taking into account the resources used and the achieved results—will probably show possibilities for improvement. If these findings are used in subsequent projects, continuous process improvement is achieved. Detailed hints for analysis and improvement of the test processes can be found in [Pol 98] and [Black 03].

Archiving testware

A further closure activity is the "conservation" of the testware for the future. Software systems are used for a long time. During this time, failures not found during testing will occur. Additionally, customers require changes. Both of these lead to changes to the program, and the changed program must be tested in every case. A major part of the test effort during →maintenance can be avoided if the testware (test cases, test logs, test infrastructure, tools, etc.) is still available. The testware should be delivered to the organization responsible for maintenance. It can then be adapted instead of being constructed from scratch, and it can also be successfully used for projects having similar requirements, after adaptation. The test material needs to be archived. Sometimes this is necessary in order to provide legal evidence of the testing done.

2.3 The Psychology of Testing

Errare humanum est

People make mistakes, but they do not like to admit them! One goal of testing software is to find discrepancies between the software and the specifications, or customer needs. The failures found must be reported to the developers. This section describes how the psychological problems occurring in connection with this can be dealt with.

The tasks of developing software are often seen as constructive actions. The tasks of examining documents and software are seen as destructive actions. The attitudes of those involved relating to their job often differ due to this perception. But these differences are not justifiable, because "testing is an extremely creative and intellectually challenging task" [Myers 79, p.15].

Developer test

"Can the developer test his own program?" is an important and frequently asked question. There is no universally valid answer. If the tester is also the author of the program, she must examine her own work very

critically. Only very few people are able to keep the necessary distance to a self-created product. Who really likes to detect and show their own mistakes? Developers would rather not find any defects in their own program text.

The main weakness of developer tests is that developers who have to test their own programs will tend to be too optimistic. There is the danger of forgetting reasonable test cases or, because they are more interested in programming than in testing, only testing superficially.

Blindness to one's own mistakes

If a developer implemented a fundamental design error—for example, if she misunderstood the task—then she will not find this using her own tests. The proper test case will not even come to mind. One possibility to decrease this problem of "blindness to one's own errors" is to work together in pairs and let a colleague test the programs.

On the other hand, it is advantageous to have a deep knowledge of one's own test object. Time is saved because it is not necessary to learn the test object. Management has to decide when saving time is an advantage over blindness to one's own errors. This must be decided depending on the criticality of the test object and the associated failure risk.

Independent test team

An independent testing team is beneficial for test quality and comprehensiveness. Further information on the formation of independent test teams can be found in section 6.1.1. The tester can look at the test object without bias. It is not the tester's own product, and the tester does not necessarily share possible developer assumptions and misunderstandings. The tester must, however, acquire the necessary knowledge about the test object in order to create test cases, which takes time. But the tester typically has more testing knowledge. A developer does not have this knowledge and must acquire it (or rather should have acquired it before, because the necessary time is often not unavailable during the project).

Failure reporting

The tester must report the failures and discrepancies observed to the author and/or to management. The way this reporting is done can contribute to cooperation between developers and testers. If it's not done well, it may negatively influence the important communication of these two groups. To prove other people's mistakes is not an easy job and requires diplomacy and tact.

Often, failures found during testing are not reproducible in the development environment for the developers. Thus, in addition to a detailed description of failures, the test environment must be documented in detail

so that differences in the environments can be detected, which can be the cause for the different behavior.

It must be defined in advance what constitutes a failure or discrepancy. If it is not clearly visible from the requirements or specifications, the customer, or management, is asked to make a decision. A discussion between the involved staff, developer, and tester as to whether this is a fault or not is not helpful. The often heard reaction of developers against any critique is, “It’s not a bug, it’s a feature!” That’s not helpful either.

Mutual comprehension

Mutual knowledge of their respective tasks improves cooperation between tester and developer. Developers should know the basics of testing and testers should have a basic knowledge of software development. This eases the understanding of the mutual tasks and problems.

The conflicts between developer and tester exist in a similar way at the management level. The test manager must report the →test results to the project manager and is thus often the messenger bringing bad news. The project manager then must decide whether there still is a chance to meet the deadline and possibly deliver software with known problems or if delivery should be delayed and additional time used for corrections. This decision depends on the severity of the failures and the possibility to work around the faults in the software.

2.4 General Principles of Testing

During the last 40 years, several principles for testing have become accepted as general rules for test work.

Principle 1:

Testing shows the presence of defects, not their absence.

Testing can show that the product fails, i.e., that there are defects. Testing cannot prove that a program is defect free. Adequate testing reduces the probability that hidden defects are present in the test object. Even if no failures are found during testing, this is no proof that there are no defects.

Principle 2:
Exhaustive testing is impossible.

It's impossible to run an exhaustive test that includes all possible values for all inputs and their combinations combined with all different preconditions. Software, in normal practice, would require an "astronomically" high number of test cases. Every test is just a sample. The test effort must therefore be controlled, taking into account risk and priorities.

Principle 3:
Testing activities should start as early as possible.

Testing activities should start as early as possible in the software life cycle and focus on defined goals. This contributes to finding defects early.

Principle 4:
Defect clustering.

Defects are not evenly distributed; they cluster together. Most defects are found in a few parts of the test object. Thus if many defects are detected in one place, there are normally more defects nearby. During testing, one must react flexibly to this principle.

Principle 5:
The pesticide paradox.

Insects and bacteria become resistant to pesticides. Similarly, if the same tests are repeated over and over, they tend to loose their effectiveness: they don't discover new defects. Old or new defects might be in program parts not executed by the test cases. To maintain the effectiveness of tests and to fight this "pesticide paradox," new and modified test cases should be developed and added to the test. Parts of the software not yet tested, or previously unused input combinations will then become involved and more defects may be found.

Principle 6:
Testing is context dependent.

Testing must be adapted to the risks inherent in the use and environment of the application. Therefore, no two systems should be tested in the exactly same way. The intensity of testing, test exit criteria, etc. should be decided upon individually for every software system, depending on its usage environment. For example, safety-critical systems require different tests than e-commerce applications.

Principle 7:
No failures means the system is useful is a fallacy.

Finding failures and repairing defects does not guarantee that the system meets user expectations and needs. Early involvement of the users in the development process and the use of prototypes are preventive measures intended to avoid this problem.

2.5 Ethical Guidelines

This section presents the Code of Tester Ethics as presented in the ISTQB Foundation Syllabus of 2011.

Dealing with critical information

Testers often have access to confidential and privileged information. This may be real, not scrambled production data used as a basis for test data, or it may be productivity data about employees. Such data or documents must be handled appropriately and must not get into the wrong hands or be misused.

For other aspects of testing work, moral or ethical rules can be applicable as well. ISTQB has based its code of ethics on the ethics from the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE). The ISTQB code of ethics[12] is as follows:

12. See [URL: ACM Ethics] and [URL: IEEE Ethics]. The guidelines listed here are from the ISTQB curriculum.

- *PUBLIC*
  »Certified software testers shall act consistently with the public interest.«
- *CLIENT AND EMPLOYER*
  »Certified software testers shall act in a manner that is in the best interest of their client and employer, consistent with the public interest.«
- *PRODUCT*
  »Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible.«
- *JUDGMENT*
  »Certified software testers shall maintain integrity and independence in their professional judgment.«
- *MANAGEMENT*
  »Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.«
- *PROFESSION*
  »Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.«
- *COLLEAGUES*
  »Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers.«
- *SELF*
  »Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.«

Ethical codes are meant to enhance public discussion about certain questions and values. Ideally, they serve as a guideline for individual responsible action. They state a "moral obligation," not a legal one. Certified testers must know the ISTQB code of ethics, which serve as a guide for daily work.

2.6 Summary

- Technical terms in the domain of software testing are often defined and used very differently, which can result in misunderstanding. Knowledge of the standards (e.g., [BS 7925-1], [IEEE 610.12], [ISO 9126]) and terminology associated with software testing is therefore an important part of the education of the Certified Tester. This book's glossary compiles the relevant terms.

- Tests are important tasks for →quality assurance in software development. The international standard ISO 9126-1 [ISO 9126] defines appropriate quality characteristics.
- The fundamental test process consists of the following phases: planning and control, analysis and design, implementation and execution, evaluation of exit criteria and reporting, and test closure activities. A test can be finished when previously defined exit criteria are fulfilled.
- A test case consists of input, expected results, and the list of defined preconditions under which the test case must run as well as the specified →postconditions. When the test case is executed, the test object shows a certain behavior. If the expected result and actual result differ, there is a failure. The expected results should be defined before test execution and during test specification (using a test oracle).
- People make mistakes, but they do not like to admit them! Because of this, psychological aspects play an important role in testing.
- The seven principles for testing must always be kept in mind during testing.
- Certified testers should know the ISTQB's ethical guidelines, which are helpful in the course of their daily work.

3 Testing in the Software Life Cycle

This chapter explains the role of testing in the entire life cycle of a software system, using the general V-model as a reference. Furthermore, we look at test levels and the test types that are used during development.

Each project in software development should be planned and executed using a life cycle model chosen in advance. Some important models were presented and explained in section 2.2. Each of these models implies certain views on software testing. From the viewpoint of testing, the general V-model according to [Boehm 79] plays an especially important role.

The role of testing within life cycle models

The V-model shows that testing activities are as valuable as development and programming. This has had a lasting influence on the appreciation of software testing. Not only every tester but every developer as well should know this general V-model and the views on testing it implies. Even if a different development model is used on a project, the principles presented in the following sections can be transferred and applied.

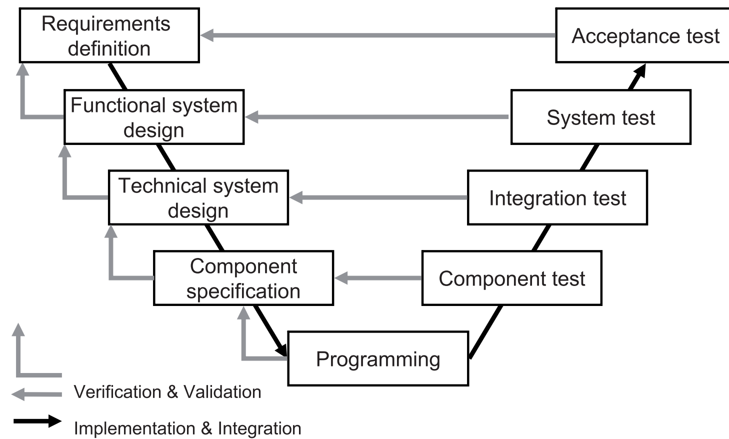
3.1 The General V-Model

The main idea behind the general V-model is that development and testing tasks are corresponding activities of equal importance. The two branches of the *V* symbolize this.

The left branch represents the development process. During development, the system is gradually being designed and finally programmed. The right branch represents the integration and testing process; the program elements are successively being assembled to form larger subsystems (integration), and their functionality is tested. →Integration and testing end when the acceptance test of the entire system has been completed. Figure 3-1 shows such a V-model.[1]

1. The V-model is used in many different versions. The names and the number of levels vary in literature and the enterprises using it.

Figure 3–1
The general V-model



The constructive activities of the left branch are the activities known from the waterfall model:

- **→Requirements definition**
  The needs and requirements of the customer or the future system user are gathered, specified, and approved. Thus, the purpose of the system and the desired characteristics are defined.
- **Functional system design**
  This step maps requirements onto functions and dialogues of the new system.
- **Technical system design**
  This step designs the implementation of the system. This includes the definition of interfaces to the system environment and decomposing the system into smaller, understandable subsystems (system architecture). Each subsystem can then be developed as independently as possible.
- **Component specification**
  This step defines each subsystem, including its task, behavior, inner structure, and interfaces to other subsystems.
- **Programming**
  Each specified component (module, unit, class) is coded in a programming language.

Through these construction levels, the software system is described in more and more detail. Mistakes can most easily be found at the abstraction level where they occurred.

Thus, for each specification and construction level, the right branch of the V-model defines a corresponding test level:

- **Component test**
  (see section 3.2) verifies whether each software →component correctly fulfills its specification.
- **→Integration test**
  (see section 3.3) checks if groups of components interact in the way that is specified by the technical system design.
- **System test**
  (see section 3.4) verifies whether the system as a whole meets the specified requirements.
- **→Acceptance test**
  (see section 3.5) checks if the system meets the customer requirements, as specified in the contract and/or if the system meets user needs and expectations.

Within each test level, the tester must make sure the outcomes of development meet the requirements that are relevant or specified on this specific level of abstraction. This process of checking the development results according to their original requirements is called →validation.

Does a product solve the intended task?

When validating,[2] the tester judges whether a (partial) product really solves the specified task and whether it is fit or suitable for its intended use.

Is it the right system?

The tester investigates to see if the system makes sense in the context of intended product use.

Does a product fulfill its specification?

In addition to validation testing, the V-model requires verification[3] testing. Unlike validation, →verification refers to only one single phase of the development process. Verification shall assure that the outcome of a particular development level has been achieved correctly and completely, according to its specification (the input documents for that development level).

Is the system correctly built?

Verification activities examine whether specifications are correctly implemented and whether the product meets its specification, but not whether the resulting product is suitable for its intended use.

2. To validate: to affirm, to declare as valid, to check if something is valid.
3. To verify: to prove, to inspect.

In practice, every test contains both aspects. On higher test levels the validation part increases. To summarize, we again list the most important characteristics and ideas behind the general V-model:

Characteristics of the general V-model

- Implementation and testing activities are separated but are equally important (left side / right side).
- The *V* illustrates the testing aspects of verification and validation.
- We distinguish between different test levels, where each test level is testing "against" its corresponding development level.

The V-model may give the impression that testing starts relatively late, after system implementation, but this is not the case. The test levels on the right branch of the model should be interpreted as levels of test execution. Test preparation (test planning, test analysis and design) starts earlier and is performed in parallel to the development phases on the left branch[4] (not explicitly shown in the V-model).

The differentiation of test levels in the V-model is more than a temporal subdivision of testing activities. It is instead defining technically very different test levels; they have different objectives and thus need different methods and tools and require personnel with different knowledge and skills. The exact contents and the process for each test level are explained in the following sections.

3.2 Component Test

3.2.1 Explanation of Terms

Within the first test level (component testing), the software units are tested systematically for the first time. The units have been implemented in the programming phase just before component testing in the V-model.

Depending on the programming language the developers used, these software units may be called by different names, such as, for example, modules and units. In object-oriented programming, they are called classes. The respective tests, therefore, are called →module tests, →unit tests (see [IEEE 1008]), and →class tests.

Component and component test

Generally, we speak of software units or components. Testing of a single software component is therefore called component testing.

4. The so-called W-model (see [Spillner 00]) is a more detailed model that explicitly shows this parallelism of development and testing.

Test basis

Component testing is based on component requirements, and the component design (or detailed design). If white box test cases will be developed or white box →test coverage will be measured, the source code can also be analyzed. However, the component behavior must be compared with the component specification.

3.2.2 Test objects

Typical test objects are program modules/units or classes, (database) scripts, and other software components. The main characteristic of component testing is that the software components are tested individually and isolated from all other software components of the system. The isolation is necessary to prevent external influences on components. If testing detects a problem, it is definitely a problem originating from the component under test itself.

Component test examines component internal aspects

The component under test may also be a unit composed of several other components. But remember that aspects internal to the components are examined, not the components' interaction with neighboring components. The latter is a task for integration tests.

Component tests may also comprise data conversion and migration components. Test objects may even be configuration data and database components.

3.2.3 Test Environment

Component testing as the lowest test level deals with test objects coming "right from the developer's desk." It is obvious that in this test level there is close cooperation with development.

Example: Testing of a class method

In the VSR subsystem *DreamCar*, the specification for calculating the price of the car states the following:

- The starting point is `baseprice` minus `discount`, where `baseprice` is the general basic price of the vehicle and `discount` is the discount to this price granted by the dealer.
- A price (`specialprice`) for a special model and the price for extra equipment items (`extraprice`) shall be added.
- If three or more extra equipment items (which are not part of the special model chosen) are chosen (`extras`), there is a discount of 10 percent on these particular items. If five or more special equipment items are chosen, this discount is increased to 15 percent.

- The discount that is granted by the dealer applies only to the `baseprice`, whereas the discount on special items applies to the special items only. These discounts cannot be combined for everything.

The following C++-function calculates the total price:[5]

```
double calculate_price
   (double baseprice, double specialprice,
   double extraprice, int extras, double discount)
{
   double addon_discount;
   double result;

   if (extras >= 3) addon_discount = 10;
   else if (extras >= 5) addon_discount = 15;
   else addon_discount = 0;
   if (discount > addon_discount)
      addon_discount = discount;

   result = baseprice/100.0*(100-discount)
   + specialprice
   + extraprice/100.0*(100-addon_discount);
   return result;
}
```

In order to test the price calculation, the tester uses the corresponding class interface calling the function `calculate_price()` with appropriate parameters and data. Then the tester records the function's reaction to the function call. That means reading and recording the return value of the previous function call. For that, a →test driver is necessary. A test driver is a program that calls the component under test and then receives the test object's reaction.

For the test object `calculate_price()`, a very simple test driver could look like this:

```
bool test_calculate_price() {

double price;
bool test_ok = TRUE;

// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (abs (price-12900.00) < 0.01);[6]
```

5. Actually, there is a defect in this program: Discount calculation for >= 5 is not reachable. The defect is used when explaining the use of white box analysis in chapter 5.
6. Floating point numbers should not be directly compared, as there may be imprecise rounding. As the result for price can be less than 12900.00, the absolute value of the difference of "price" and 12900.00 must be evaluated.

```
// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (abs (price-34050.00) < 0.01);

// testcase ...

// test result
return test_ok;
}
```

The preceding test driver is programmed in a very simple way. Some useful extensions could be, for example, a facility to record the test data and the results, including date and time of the test, or a function that reads test cases from a table, file, or database.

To write test drivers, programming skills and knowledge of the component under test are necessary. The component's program code must be available. The tester must understand the test object (in the example, a class function) so that the call of the test object can be correctly programmed in the test driver. To write a suitable test driver, the tester must know the programming language and suitable programming tools must be available.

This is why the developers themselves usually perform the component testing. Although this is truly a component test, it may also be called developer test. The disadvantages of a programmer testing his own program were discussed in section 2.3.

Often, component testing is also confused with debugging. But debugging is not testing. Debugging is finding the cause of failures and removing them, while testing is the systematic approach for finding failures.

Hint

- Use of component testing frameworks (see [URL: xunit]) reduces the effort involved in programming test drivers and helps to standardize a project's component testing architecture. [Vigenschow 2010] demonstrates the use of these frameworks using examples of Junit for Java as well as nUnit and CppUnit for C++. Generic test drivers make it easier to use colleagues[7] who are not familiar with all details of the particular component and the programming environment for testing. Such test drivers can, for example, be used through a command interface and provide comfortable mechanisms for managing the test data and for recording and analyzing the tests. As all test data and test protocols are structured in a very similar way, this enables analysis of the tests across several components.

7. Sometimes, two programmers work together, each of them testing the components that their colleague has developed. This is called *buddy testing* or *code swaps*.

3.2.4 Test objectives

The test level called component test is not only characterized by the kind of test objects and the testing environment, the tester also pursues test objectives that are specific for this phase.

Testing the functionality

The most important task of component testing is to check that the entire functionality of the test object works correctly and completely as required by its specification (see →functional testing). Here, *functionality* means the input/output behavior of the test object. To check the correctness and completeness of the implementation, the component is tested with a series of test cases, where each test case covers a particular input/output combination (partial functionality).

Example: Test of the VSR price calculation

The test cases for the price calculation of *DreamCar* in the previous example very clearly show how the examination of the input/output behavior works. Each test case calls the test object with a particular combination of data; in this example, the price for the vehicle in combination with a different set of extra equipment items. It is then examined to see whether the test object, given this input data, calculates the correct price. For example, test case 2 checks the partial functionality of "discount with five or more special equipment items." If test case 2 is executed, we can see that the test object calculates the wrong total price. Test case 2 produces a failure. The test object does not completely meet the functional requirements.

Typical software defects found during functional component testing are incorrect calculations or missing or wrongly chosen program paths (e.g., special cases that were forgotten or misinterpreted).

Later, when the whole system is integrated, each software component must be able to cooperate with many neighboring components and exchange data with them. A component may then possibly be called or used in a wrong way, i.e., not in accordance with its specification. In such cases, the wrongly used component should not just suspend its service or cause the whole system to crash. Rather, it should be able to handle the situation in a reasonable and robust way.

Testing robustness

This is why testing for →robustness is another very important aspect of component testing. The way to do this is the same as in functional testing. However, the test focuses on items either not allowed or forgotten in the specification. The tests are function calls, test data, and special cases. Such test cases are also called →negative tests. The component's reaction should be an appropriate exception handling. If there is no such exception

handling, wrong inputs can trigger domain faults like division by zero or access to a null pointer. Such faults could lead to a program crash.

Example: Negative test

In the price calculation example, such negative tests are function calls with negative values, values that are far too large, or wrong data types (for example, `char` instead of `int`):[8]

```
// testcase 20
price = calculate_price(-1000.00,0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_PRICE);
…
// testcase 30
price = calculate_price("abc",0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_ARGUMENT);
```

Excursion

Some interesting aspects become clear:

- There are at least as many reasonable negative tests as positive ones.
- The test driver must be extended in order to be able to evaluate the test object's exception handling.
- The test object's exception handling (the analysis of `ERR_CODE` in the previous example) requires additional functionality. Often more than 50% of the program code deals with exception handling. Robustness has its cost.

Component testing should not only check functionality and robustness.

All the component's characteristics that have a crucial influence on its quality and that cannot be tested in higher test levels (or only with a much higher cost) should be checked during component testing. This may be nonfunctional characteristics like efficiency[9] and maintainability.

Efficiency test

Efficiency refers to how efficiently the component uses computer resources. Here we have various aspects such as use of memory, computing time, disk or network access time, and the time required to execute the component's functions and algorithms. In contrast to most other nonfunctional tests, a test object's efficiency can be measured during the test. Suitable criteria are measured exactly (e.g., memory usage in kilobytes, response times in milliseconds). Efficiency tests are seldom performed for all the components of a system. Efficiency is usually only verified in effi-

8. Depending on the compiler, data type errors can be detected during the compiling process.
9. The opportunity to use these types of checks on a component level instead of during a system test is not often exploited. This leads to efficiency problems only becoming visible shortly before the planned release date. Such problems can then only be corrected or attenuated at significant cost.

ciency-critical parts of the system or if efficiency requirements are explicitly stated by specifications. This happens, for example, in testing embedded software, where only limited hardware resources are available. Another example is testing real-time systems, where it must be guaranteed that the system follows given timing constraints.

Maintainability test

A maintainability test includes all the characteristics of a program that have an influence on how easy or how difficult it is to change the program or to continue developing it. Here, it is crucial that the developer fully understands the program and its context. This includes the developer of the original program who is asked to continue development after months or years as well as the programmer who takes over responsibility for a colleague's code. The following aspects are most important for testing maintainability: code structure, modularity, quality of the comments in the code, adherence to standards, understandability, and currency of the documentation.

Example: Code that is difficult to maintain

The code in the example `calculate_price()` is not good enough. There are no comments, and numeric constants are not declared but are just written into the code. If such a value must be changed later, it is not clear whether and where this value occurs in other parts of the system, nor is it clear how to find and change it.

Of course, such characteristics cannot be tested by →dynamic tests (see chapter 5). Analysis of the program text and the specifications is necessary. →Static testing, and especially reviews (see section 4.1) are the correct means for that purpose. However, it is best to include such analyses in the component test because the characteristics of a single component are examined.

3.2.5 Test Strategy

As we explained earlier, component testing is very closely related to development. The tester usually has access to the source code, which makes component testing the domain of white box testing (see section 5.2).

White box test

The tester can design test cases using her knowledge about the component's program structures, functions, and variables. Access to the program code can also be helpful for executing the tests. With the help of special tools (→debugger, see section 7.1.4), it is possible to observe program variables during test execution. This helps in checking for correct or incorrect behavior of the component. The internal state of a component

cannot only be observed; it can even be manipulated with the debugger. This is especially useful for robustness tests because the tester is able to trigger special exceptional situations.

Example: Code as test basis

Analyzing the code of `calculate_price()`, the following command can be recognized as a line that is relevant for testing:

```
if (discount > addon_discount)
      addon_discount = discount;
```

Additional test cases that lead to fulfilling the condition (`discount > addon_discount`) can easily be derived from the code. The specification of the price calculation contains no information about this situation; the implemented functionality is extra: it is not supposed to be there.

In reality, however, component testing is often done as a pure black box testing, which means that the code structure is not used to design test cases.[10] On the one hand, real software systems consist of countless elementary components; therefore, code analysis for designing test cases is probably only feasible with very few selected components.

On the other hand, the elementary components will later be integrated into larger units. Often, the tester only recognizes these larger units as units that can be tested, even in component testing. Then again, these units are already too large to make observations and interventions on the code level with reasonable effort. Therefore, integration and testing planning must answer the question of whether to test elementary parts or only larger units during component testing.

"Test first" development

Test first programming is a modern approach in component testing. The idea is to design and automate the tests first and program the desired component afterwards.

This approach is very iterative. The program code is tested with the available test cases. The code is improved until it passes the tests. This is also called test-driven development (see [Link 03]).

10. This is a serious flaw because 60 to 80% of the code often is never executed—a perfect hideout for bugs.

3.3 Integration Test

3.3.1 Explanation of Terms

After the component test, the second test level in the V-model is integration testing. A precondition for integration testing is that the test objects subjected to it (i.e., components) have already been tested. Defects should, if possible, already have been corrected.

Integration

Developers, testers, or special integration teams then compose groups of these components to form larger structural units and subsystems. This connecting of components is called integration.

Integration test

Then the structural units and subsystems must be tested to make sure all components collaborate correctly. Thus, the goal of the integration test is to expose faults in the interfaces and in the interaction between integrated components.

Test basis

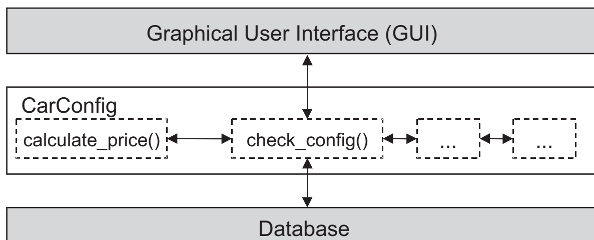
The test basis may be the software and system design or system architecture, or workflows through several interfaces and use cases.

Why is integration testing necessary if each individual component has already been tested? The following example illustrates the problem.

Example: Integration test VSR-DreamCar

The VSR subsystem *DreamCar* (see figure 2-1) consists of several elementary components.

Figure 3–2
Structure of the subsystem VSR-DreamCar



One element is the class `CarConfig` with the methods `calculate_price()`, `check_config()`, and other methods. `check_config()` retrieves all the vehicle data from a database and presents them to the user through a graphical user interface (GUI). From the user's point of view, this looks like figure 3-3.

When the user has chosen the configuration of a car, `check_config()` executes a plausibility check of the configuration (base model of the vehicle, special equipment, list of further extra items) and then calculates the price. In this example (see figure 3-3), the total resulting price from the base model of the chosen vehicle, the special model, and the extra equipment should be \$29,000 + \$1,413 + \$900 = \$31,313.

However, the price indicated is only $30,413. Obviously, in the current program version, accessories (e.g., alloy rims) can be selected without paying for them. Somewhere between the GUI and `calculate_price()`, the fact that alloy rims were chosen gets lost.

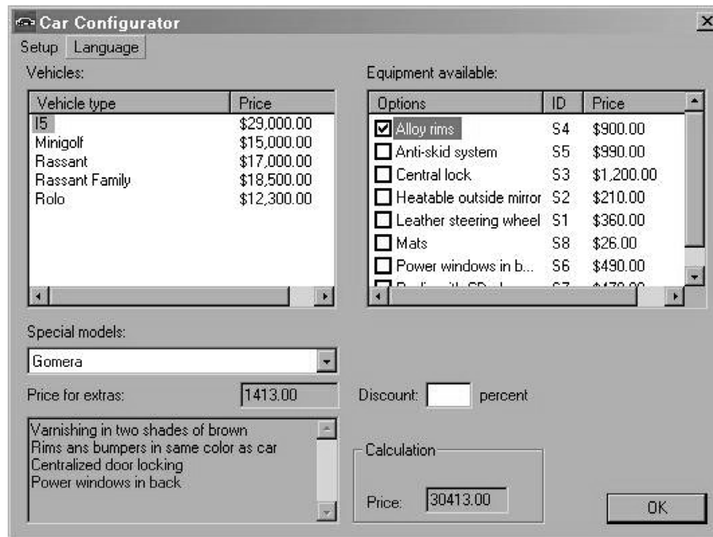


Figure 3–3
User interface for the VSR subsystem DreamCar

If the test protocols of the previous component tests show that the fault is neither in the function `calculate_price()` nor in `check_config()`, the cause of the problem could be a faulty data transmission between the GUI and `check_config()` or between `check_config()` and `calculate_price()`.

Even if a complete component test had been executed earlier, such interface problems can still occur. Because of this, integration testing is necessary as a further test level. Its task is to find collaboration and interoperability problems and isolate their causes.

Example: VSR integration test

Integration of the single components to the subsystem *DreamCar* is just the beginning of the integration test in the project VSR. The other subsystems of the VSR (see chapter 2, figure 2-1) must also be integrated. Then, the subsystems must be connected to each other. *DreamCar* has to be connected to the subsystem *ContractBase*, which is connected to the subsystems *JustInTime* (order management), *NoRisk* (vehicle insurance), and *EasyFinance* (financing). In one of the last steps of integration, VSR is connected to the external mainframe in the IT center of the enterprise.

Integration testing in the large

As the example shows, interfaces to the system environment (i.e., external systems) are also subject to integration and integration testing. When interfaces to external software systems are examined, we sometimes speak of →system integration testing, higher-level integration testing, or integration testing in the large (integration of components is then integration test in the small, sometimes called →component integration testing). System integration testing can be executed only after system testing. The development team has only one-half of such an external interface under its control. This constitutes a special risk. The other half of the interface is determined by an external system. It must be taken as it is, but it is subject to unexpected change. Passing a system integration test is no guarantee that the system will function flawlessly in the future.

Integration levels

Thus, there may be several integration levels for test objects of different sizes. Component integration tests will test the interfaces between internal components or between internal subsystems. System integration tests focus on testing interfaces between different systems and between hardware and software. For example, if business processes are implemented as a workflow through several interfacing systems and problems occur, it may be very expensive and challenging to find the defect in a special component or interface.

3.3.2 Test objects

Assembled components

Step-by-step, during integration, the different components are combined to form larger units (see section 3.3.5). Ideally, there should be an integration test after each of these steps. Each subsystem may then be the basis for integrating further larger units. Such units (subsystems) may be test objects for the integration test later.

External systems or acquired components

In reality, a software system is seldom developed from scratch. Usually, an existing system is changed, extended, or linked to other systems (for example database systems, networks, new hardware). Furthermore, many system components are →commercial off-the-shelf (COTS) software products (for example, the database in *DreamCar*). In component testing, such existing or standard components are probably not tested. In the integration test, however, these system components must be taken into account and their collaboration with other components must be examined.

The most important test objects of integration testing are internal interfaces between components. Integration testing may also comprise configuration programs and configuration data. Finally, integration or

system integration testing examines subsystems for correct database access and correct use of other infrastructure components.

3.3.3 The Test Environment

As with component testing, test drivers are needed in the integration test. They send test data to the test objects, and they receive and log the results. Because the test objects are assembled components that have no interfaces to the "outside" other than their constituting components, it is obvious and sensible to reuse the available test drivers for component testing.

Reuse of the test environment

If the component test was well organized, then some test drivers should be available. It could be one generic test driver for all components or at least test drivers that were designed with a common architecture and are compatible with each other. In this case, the testers can reuse these test drivers without much effort.

If a component test is poorly organized, there may be usable test drivers for only a few of the components. Their user interface may also be completely different, which will create trouble. During integration testing in a much later stage of the project, the tester will need to put a lot of effort into the creation, change, or repair of the test environment. This means that valuable time needed for test execution is lost.

Monitors are necessary

During integration testing, additional tools, called monitors, are required. →Monitors are programs that read and log data traffic between components. Monitors for standard protocols (e.g., network protocols) are commercially available. Special monitors must be developed for the observation of project-specific component interfaces.

3.3.4 Test objectives

Wrong interface formats

The test objectives of the test level integration test are clear: to reveal interface problems as well as conflicts between integrated parts.

Problems can arise when attempting to integrate two single components. For example, their interface formats may not be compatible with each other because some files are missing or because the developers have split the system into completely different components than specified (chapter 4 covers static testing, which may help finding such issues).

Typical faults in data exchange

The harder-to-find problems, however, are due to the execution of the connected program parts. These kinds of problems can only be found by dynamic testing. They are faults in the data exchange or in the communication between the components, as in the following examples:

- A component transmits syntactically incorrect or no data. The receiving component cannot operate or crashes (functional fault in a component, incompatible interface formats, protocol faults).
- The communication works but the involved components interpret the received data differently (functional fault of a component, contradicting or misinterpreted specifications).
- Data is transmitted correctly but at the wrong time, or it is late (timing problem), or the intervals between the transmissions are too short (throughput, load, or capacity problem).

Example: Integration problems in VSR

The following interface failures could occur during the VSR integration test. These can be attributed to the previously mentioned failure types:

- In the GUI of the *DreamCar* subsystem, selected extra equipment items are not passed on to `check_config()`. Therefore, the price and the order data would be wrong.
- In *DreamCar,* a certain code number (e.g., 442 for metallic blue) represents the color of the car. In the order management system running on the external mainframe, however, some code numbers are interpreted differently (there, for example, 442 may represent red). An order from the VSR, seen there as correct, would lead to delivery of the wrong product.
- The mainframe computer confirms an order after checking whether delivery would be possible. In some cases, this examination takes so long that the VSR assumes a transmission failure and aborts the order. A customer who has carefully chosen her car would not be able to order it.

None of these failures can be found in the component test because the resulting failures occur only in the interaction between two software components.

Nonfunctional tests may also be executed during integration testing, if attributes mentioned below are important or are considered at risk. These attributes may include reliability, performance, and capacity.

Can the component test be omitted?

Is it possible to do without the component test and execute all the test cases after integration is finished? Of course, this is possible, and in practice it is regretfully often done, but only at the risk of great disadvantages:

- Most of the failures that will occur in a test designed like this are caused by functional faults within the individual components. An implicit component test is therefore carried out, but in an environment that is not suitable and that makes it harder to access the individual components.

- Because there is no suitable access to the individual component, some failures cannot be provoked and many faults, therefore, cannot be found.
- If a failure occurs in the test, it can be difficult or impossible to locate its origin and to isolate its cause.

The cost of trying to save effort by cutting the component test is finding fewer of the existing faults and experiencing more difficulty in diagnosis. Combining a component test with a subsequent integration test is more effective and efficient.

3.3.5 Integration Strategies

In which order should the components be integrated in order to execute the necessary test work as efficiently—that is, as quickly and easily—as possible? Efficiency is the relation between the cost of testing (the cost of test personnel and tools, etc.) and the benefit of testing (number and severity of the problems found) in a certain test level.

The test manager has to decide this and choose and implement an optimal integration strategy for the project.

Components are completed at different times

In practice, different software components are completed at different times, weeks or even months apart. No project manager or test manager can allow testers to sit around and do nothing while waiting until all the components are developed and they are ready to be integrated.

An obvious ad hoc strategy to quickly solve this problem is to integrate the components in the order in which they are ready. This means that as soon as a component has passed the component test, it is checked to see if it fits with another already tested component or if it fits into a partially integrated subsystem. If so, both parts are integrated and the integration test between both of them is executed.

Example: Integration Strategy in the VSR project

In the VSR project, the central subsystem *ContractBase* turns out to be more complex than expected. Its completion is delayed for several weeks because the work on it costs much more than originally expected. To avoid losing even more time, the project manager decides to start the tests with the available components *DreamCar* and *NoRisk*. These do not have a common interface, but they exchange data through *ContractBase*. To calculate the price of the insurance, *NoRisk* needs to know which type of vehicle was chosen because this determines the price and other parameters of the insurance. As a temporary replacement for *ContractBase*, a →stub is programmed. The stub receives simple car configuration data from *DreamCar*, then determines the vehicle type code from this data and passes it on

to NoRisk. Furthermore, the stub makes it possible to put in different relevant data about the customer. *NoRisk* calculates the insurance price from the data and shows it in a window so it can be checked. The price is also saved in a test log. The stub serves as a temporary replacement for the still missing subsystem *ContractBase*.

This example makes it clear that the earlier the integration test is started (in order to save time), the more effort it will take to program the stubs. The test manager has to choose an integration strategy in order to optimize both factors (time savings vs. cost for the testing environment).

Constraints for integration

Which strategy is optimal (the most timesaving and least costly strategy) depends on the individual circumstances in each project. The following items must be analyzed:

- The **system architecture** determines how many and which components the entire system consists of and in which way they depend on each other.
- The **project plan** determines at what time during the course of the project the parts of the system are developed and when they should be ready for testing. The test manager should be consulted when determining the order of implementation.
- The **test plan** determines which aspects of the system shall be tested, how intensely, and on which test level this has to happen.

Discuss the integration strategy

The test manager, taking into account these general constraints, has to design an optimal integration strategy for the project. Because the integration strategy depends on delivery dates, the test manager should consult the project manager during project planning. The order of component implementation should be suitable for integration testing.

Generic strategies

When making plans, the test manager can follow these generic integration strategies:

- **Top-down integration**
  The test starts with the top-level component of the system that calls other components but is not called itself (except for a call from the operating system). Stubs replace all subordinate components. Successively, integration proceeds with lower-level components. The higher level that has already been tested serves as test driver.
  - Advantage: Test drivers are not needed, or only simple ones are required, because the higher-level components that have already been tested serve as the main part of the test environment.

- Disadvantage: Stubs must replace lower-level components not yet integrated. This can be very costly.

- **Bottom-up integration**
  The test starts with the elementary system components that do not call further components, except for functions of the operating system. Larger subsystems are assembled from the tested components and then tested.
  - Advantage: No stubs are needed.
  - Disadvantage: Test drivers must simulate higher-level components.

- **Ad hoc integration**
  The components are being integrated in the (casual) order in which they are finished.
  - Advantage: This saves time because every component is integrated as early as possible into its environment.
  - Disadvantage: Stubs as well as test drivers are required.

- **Backbone integration**
  A skeleton or backbone is built and components are gradually integrated into it [Beizer 90].
  - Advantage: Components can be integrated in any order.
  - Disadvantage: A possibly labor-intensive skeleton or backbone is required.

Top-down and Bottom-up integration in their pure form can be applied only to program systems that are structured in a strictly hierarchical way; in reality, this rarely occurs. This is the reason a more or less individualized mix of the previously mentioned integration strategies[11] might be chosen.

Avoid the big bang!

Any nonincremental integration—also called →big bang integration—should be avoided. Big bang integration means waiting until all software elements are developed and then throwing everything together in one step. This typically happens due to the lack of an integration strategy. In the worst cases, even component testing is skipped. There are obvious disadvantages of this approach:

- The time leading up to the big bang is lost time that could have been spent testing. As testing always suffers from lack of time, no time that could be used for testing should be wasted.

11. Special integration strategies can be followed for object-oriented, distributed, and real-time systems (see [Winter 98], [Bashir 99], [Binder 99]).

- All the failures will occur at the same time. It will be difficult or impossible to get the system to run at all. It will be very difficult and time-consuming to localize and correct defects.

3.4 System Test

3.4.1 Explanation of Terms

After the integration test is completed, the third and next test level is the system test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after executing component and integration tests? The reasons for this are as follows:

Reasons for system test

- In the lower test levels, the testing was done against technical specifications, i.e., from the technical perspective of the software producer. The system test, though, looks at the system from the perspective of the customer and the future user.[12] The testers validate whether the requirements are completely and appropriately implemented.
- Many functions and system characteristics result from the interaction of all system components; consequently, they are visible only when the entire system is present and can be observed and tested only there.

Example: VSR-System tests

The main purpose of the VSR-System is to make ordering a car as easy as possible.

While ordering a car, the user uses all the components of the VSR-System: the car is configured (*DreamCar*), financing and insurance are calculated (*Easy-Finance*, *NoRisk*), the order is transmitted to production (*JustInTime*), and the contracts are archived (*ContractBase*). The system fulfills its purpose only when all these system functions and all the components collaborate correctly. The system test determines whether this is the case.

The test basis includes all documents or information describing the test object on a system level. This may be system requirements, specifications, risk analyses if present, user manuals, etc.

12. The customer (who has ordered and paid for the system) and the user (who uses the system) can be different groups of people or organizations with their own specific interests and requirements for the system.

3.4.2 Test Objects and Test Environment

After the completion of the integration test, the software system is complete. The system test tests the system as a whole in an environment as similar as possible to the intended →production environment.

Instead of test drivers and stubs, the hardware and software products that will be used later should be installed on the test platform (hardware, system software, device driver software, networks, external systems, etc.). Figure 3-4 shows an example of the VSR-System test environment.

The system test not only tests the system itself, it also checks system and user documentation, like system manuals, user manuals, training material, and so on. Testing configuration settings as well as optimizing the system configuration during load and performance testing (see section 3.7.2) must often be covered.

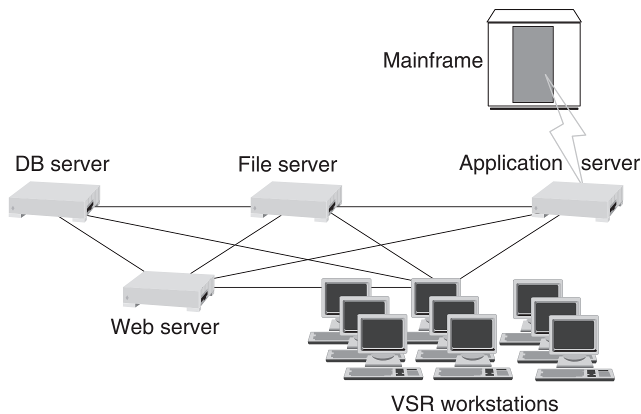


Figure 3–4
Example of a system test environment

→data quality

It is getting more and more important to check the quality of data in systems that use a database or large amounts of data. This should be included in the system test. The data itself will then be new test objects. It must be assured that it is consistent, complete, and up-to-date. For example, if a system finds and displays bus connections, the station list and schedule data must be correct.

System test requires a separate test environment

One mistake is commonly made to save costs and effort: instead of the system being tested in a separate environment, the system test is executed in the customer's operational environment. This is detrimental for a couple of reasons:

- During system testing, it is likely that failures will occur, resulting in damage to the customer's operational environment. This may lead to expensive system crashes and data loss in the production system.
- The testers have only limited or no control over parameter settings and the configuration of the operational environment. The test conditions may change over time because the other systems in the customer's environment are running simultaneously with the test. The system tests that have been executed cannot be reproduced or can only be reproduced with difficulty (see section 3.7.4 on regression testing).

System test effort is often underestimated

The effort of an adequate system test must not be underestimated, especially because of the complex test environment. [Bourne 97] states the experience that at the beginning of the system test, only half of the testing and quality control work has been done (especially when a client/server system is developed, as in the VSR-example).

3.4.3 Test Objectives

It is the goal of the system test to validate whether the complete system meets the specified functional and nonfunctional requirements (see sections 3.7.1 and 3.7.2) and how well it does that. Failures from incorrect, incomplete, or inconsistent implementation of requirements should be detected. Even undocumented or forgotten requirements should be identified.

3.4.4 Problems in System Test Practice

Excursion

In (too) many projects, the requirements are incompletely or not at all written down. The problem this poses for testers is that it's unclear how the system is supposed to behave. This makes it hard to find defects.

Unclear system requirements

If there are no requirements, then all behaviors of a system would be valid and assessment would be impossible. Of course, the users or the customers have a certain perception of what they expect of "their" software system. Thus, there *must be* requirements. Yet sometimes these requirements are not written down anywhere; they exist only in the minds of a few people who are involved in the project. The testers then have the undesirable role of gathering information about the required behavior after the fact. One possible technique to cope with such a situation is exploratory testing (see section 5.3, and for more detailed discussion, [Black 02]).

Missed decisions

While the testers identify the original requirements, they will discover that different people may have completely different views and ideas on the same subject. This is not surprising if the requirements have never been documented, reviewed, or released during the project. The consequences for those responsible for system testing are less desirable: They must collect information on the requirements; they also have to make decisions that should have been made many months earlier. This collection of information may be very costly and time consuming. Test completion and release of the completed system will surely be delayed.

Project fail

If the requirements are not specified, of course the developers do not have clear objectives either. Thus, it is very unlikely that the developed system will meet the implicit requirements of the customer. Nobody can seriously expect that it is possible to develop a usable system given these conditions. In such projects, execution of the system test can probably only announce the collapse of the project.

3.5 Acceptance Test

All the test levels described thus far represent testing activities that are under the producer's responsibility. They are executed before the software is presented to the customer or the user.

Before installing and using the software in real life (especially for software developed individually for a customer), another last test level must be executed: the acceptance test. Here, the focus is on the customer's and user's perspective. The acceptance test may be the only test that the customers are actually involved in or that they can understand. The customer may even be responsible for this test!

→Acceptance tests may also be executed as a part of lower test levels or be distributed over several test levels:

- A commercial-off-the-shelf product (COTS) can be checked for acceptance during its integration or installation.
- Usability of a component can be acceptance tested during its component test.
- Acceptance of new functionality can be checked on prototypes before system testing.

There are four typical forms of acceptance testing:

- Contract acceptance testing
- User acceptance testing
- Operational acceptance testing
- Field testing (alpha and beta testing)

How much acceptance testing?

How much acceptance testing should be done is dependent on the product risk. This may be very different. For customer-specific systems, the risk is high and a comprehensive acceptance test is necessary. At the other extreme, if a piece of standard software is introduced, it may be sufficient to install the package and test a few representative usage scenarios. If the system interfaces with other systems, collaboration of the systems through these interfaces must be tested.

Test basis

The test basis for acceptance testing can be any document describing the system from the user or customer viewpoint, such as, for example, user or system requirements, use cases, business processes, risk analyses, user process descriptions, forms, reports, and laws and regulations as well as descriptions of maintenance and system administration rules and processes.

3.5.1 Contract Acceptance Testing

If customer-specific software was developed, the customer will perform contract acceptance testing (in cooperation with the vendor). Based on the results, the customer considers whether the software system is free of (major) deficiencies and whether the service defined by the development contract has been accomplished and is acceptable. In case of internal software development, this can be a more or less formal contract between the user department and the IT department of the same enterprise.

Acceptance criteria

The test criteria are the acceptance criteria determined in the development contract. Therefore, these criteria must be stated as unambiguously as possible. Additionally, conformance to any governmental, legal, or safety regulations must be addressed here.

In practice, the software producer will have checked these criteria within his own system test. For the acceptance test, it is then enough to rerun the test cases that the contract requires as relevant for acceptance, demonstrating to the customer that the acceptance criteria of the contract have been met.

Because the supplier may have misunderstood the acceptance criteria, it is very important that the acceptance test cases are designed by or at least thoroughly reviewed by the customer.

Customer (site) acceptance test

In contrast to system testing, which takes place in the producer environment, acceptance testing is run in the customer's actual operational environment.[13] Due to these different testing environments, a test case that worked correctly during the system test may now suddenly fail. The acceptance test also checks the delivery and installation procedures. The acceptance environment should be as similar as possible to the later operational environment. A test in the operational environment itself should be avoided to minimize the risk of damage to other software systems used in production.

The same techniques used for test case design in system testing can be used to develop acceptance test cases. For administrative IT systems, business transactions for typical business periods (like a billing period) should be considered.

3.5.2 Testing for User Acceptance

Another aspect concerning acceptance as the last phase of validation is the test for user acceptance. Such a test is especially recommended if the customer and the user are different.

Example: Different user groups

In the VSR example, the responsible customer is a car manufacturer. But the car manufacturer's shops will use the system. Employees and customers who want to purchase cars will be the system's end users. In addition, some clerks in the company's headquarter will work with the system, e.g., to update price lists in the system.

Get acceptance of every user group

Different user groups usually have completely different expectations of a new system. Users may reject a system because they find it "awkward" to use, which can have a negative impact on the introduction of the system. This may happen even if the system is completely OK from a functional point of view. Thus, it is necessary to organize a user acceptance test for each user group. The customer usually organizes these tests, selecting test cases based on business processes and typical usage scenarios.

13. Sometimes the acceptance test consists of two cycles: the first in the system test environment, the second in the customer's environment.

Present prototypes to the users early

If major user acceptance problems are detected during acceptance testing, it is often too late to implement more than cosmetic countermeasures. To prevent such disasters, it is advisable to let a number of representatives from the group of future users examine prototypes of the system early.

3.5.3 Operational (Acceptance) Testing

Operational (acceptance) testing assures the acceptance of the system by the system administrators.[14] It may include testing of backup/restore cycles (including restoration of copied data), disaster recovery, user management, and checks of security vulnerabilities.

3.5.4 Field Testing

If the software is supposed to run in many different operational environments, it is very expensive or even impossible for the software producer to create a test environment for each of them during system testing. In such cases, the software producer may choose to execute a →field test after the system test. The objective of the field test is to identify influences from users' environments that are not entirely known or specified and to eliminate them if necessary. If the system is intended for the general market (a COTS system), this test is especially recommended.

Testing done by representative customers

For this purpose, the producer delivers stable prerelease versions of the software to preselected customers who adequately represent the market for this software or whose operational environments are appropriately similar to possible environments for the software.

These customers then either run test scenarios prescribed by the producer or run the product on a trial basis under realistic conditions. They give feedback to the producer about the problems they encountered along with general comments and impressions about the new product. The producer can then make the specific adjustments.

Alpha and beta testing

Such testing of preliminary versions by representative customers is also called →alpha testing or →beta testing. Alpha tests are carried out at the producer's location, while beta tests are carried out at the customer's site.

A field test should not replace an internal system test run by the producer (even if some producers do exactly this). Only when the system test

14. This verifies that the system complies with the needs of the system administrators.

has proven that the software is stable enough should the new product be given to potential customers for a field test.

Dogfood test

A new term in software testing is *dogfood test*. It refers to a kind of internal field testing where the product is distributed to and used by internal users in the company that developed the software. The idea is that "if you make dogfood, try it yourself first." Large suppliers of software like Microsoft and Google advocate this approach before beta testing.

3.6 Testing New Product Versions

Until now, it was assumed that a software development project is finished when the software passes the acceptance test and is deployed. But that's not the reality. The first deployment marks only the beginning of the software life cycle. Once it is installed, it will often be used for years or decades and is changed, updated, and extended many times. Each time that happens, a new →version of the original product is created. The following sections explain what must be considered when testing such new product versions.

3.6.1 Software Maintenance

Software does not wear out. Unlike with physical industry products, the purpose of software maintenance is not to maintain the ability to operate or to repair damages caused by use. Defects do not originate from wear and tear. They are design faults that already exist in the original version. We speak of software maintenance when a product is adapted to new operational conditions (adaptive maintenance, updates of operating systems, databases, middleware) or when defects that have been in the product before are corrected (corrective maintenance). Testing changes made during maintenance can be difficult because the system's specifications are often out of date or missing, especially in the case of legacy systems.

Example: Analysis of VSR hotline requests

The VSR-System has been distributed and installed after intense testing. In order to find areas with weaknesses that had not been found previously, the central hotline generates an analysis of all requests that have come in from the field. Here are some examples:

1. A few dealers use the system on an unsupported platform with an old version of the operating system. In such environments, sometimes the host access causes system crashes.
2. Many customers consider the selection of extra equipment to be awkward, especially when they want to compare prices between different packages of

extra equipment. Many users would therefore like to save equipment configurations and to be able to retrieve them after a change.
3. Some of the seldom-occurring insurance prices cannot be calculated at all because the corresponding calculation wasn't implemented in the insurance component.
4. Sometimes, even after more than 15 minutes, a car order is not yet confirmed by the server. The system cuts the connection after 15 minutes to avoid having unused connections remain open. The customers are angry with this because they waste a lot of time waiting in vain for confirmation of the purchase order. The dealer then has to repeat inputting the order and then has to mail the confirmation to the customer.

Problem 1 is the responsibility of the dealer because he runs the system on a platform for which it was not intended. Still, the software producer might change the program to allow it to be run on this platform to, for example, save the dealer from the cost of a hardware upgrade.

Problems like number 2 will always arise, regardless of how well and completely the requirements were originally analyzed. The new system will generate many new experiences and therefore new requirements will naturally arise.

Improve the test plan

Problem 3 could have been detected during system testing. But testing cannot guarantee that a system is completely fault free. It can only provide a sample with a certain probability to reveal failures. A good test manager will analyze which kind of testing would have detected this problem and will adequately improve or adapt the test plan.

Problem 4 had been detected in the integration test and had been solved. The VSR-System waits for a confirmation from the server for more than 15 minutes without cutting the connection. The long waiting time happens in special cases, when certain batch processes are run in the host computer. The fact that the customer does not want to wait in the shop for such a long time is another subject.

These four examples represent typical problems that will be found in even the most mature software system:

1. The system is run under new operating conditions that were not predictable and not planned.
2. The customers express new wishes.
3. Functions are necessary for rarely occurring special cases that were not anticipated.
4. Crashes that happen rarely or only after a very long run time are reported. These are often caused by external influences.

Therefore, after its deployment, every software system requires certain corrections and improvements. In this context, we speak of software mainte-

nance. But the fact that maintenance is necessary in any case must not be used as a pretext for cutting down on component, integration, or system testing. We sometime hear, “We must continuously publish updates anyway, so we don’t need to take testing so seriously, even if we miss defects.” Managers behaving this way do not understand the true costs of failures.

Testing after maintenance work

If the production environment has been changed or the system is ported to a new environment (for example, by migration to a new platform), a new acceptance test should be run by the organization responsible for operations. If data has to be migrated or converted, even this aspect must be tested for correctness and completeness.

Otherwise, the test strategy for testing a changed system is the same as for testing every new product version: Every new or changed part of the code must be tested. Additionally, in order to avoid side effects, the remainder of the system should be regression tested (see section 3.7.4) as comprehensibly as possible. The test will be easier and more successful if even maintenance releases are planned in advance and considered in the test plans.

There should be two strategies: one for emergency fixes (or “hot fixes”) and one for planned releases. For an ordinary release, a test approach should be planned early, comprising thorough testing of anything new or changed as well as regression testing. For an emergency fix, a minimal test should be executed to minimize the time to release. Then the normal comprehensive test should be executed as soon as possible afterwards.

Testing before retirement

If a system is scheduled for retirement, then some testing is also useful.

Testing for the retirement of a system should include the testing of data archiving or data migration into the future system.

3.6.2 Testing after Further Development

Apart from maintenance work necessary because of failures, there will be changes and extensions to the product that project management has intended from the beginning.

Example: Planning of the VSR development

In the development plan for VSR release 2, the following work is scheduled:

1. New communication software will be installed on the host in the car manufacturer’s computing center; therefore, the VSR communication module must be adapted.

2. Certain system extensions that could not be finished in release 1 will now be delivered in release 2.
3. The installation base shall be extended to the EU dealer network. Therefore, specific adaptations necessary for each country must be integrated and all the manuals and the user interface must be translated.

These three tasks come neither from defects nor from unforeseen user requests. So they are not part of ordinary maintenance but instead normal further product development.

The first point results from a planned change of a neighbor system. Point 2 involves functionality that had been planned from the beginning but could not be implemented as early as intended. Point 3 represents extensions that become necessary in the course of a planned market expansion.

A software product is certainly not finished with the release of the first version. Additional development is continuously occurring. An improved product version will be delivered at certain intervals, such as, e.g., once a year. It is best to synchronize these →releases with the ongoing maintenance work. For example, every six months a new version is introduced: one maintenance update and one genuine functional update.

After each release, the project effectively starts over, running through all the project phases. This approach is called iterative software development. Nowadays this is the usual way of developing software.[15]

Testing new releases

How must testing respond to this? Do we have to completely rerun all the test levels for every release of the product? Yes, if possible! As with maintenance testing, anything new or changed should be tested, and the remainder of the system should be regression tested to find unexpected side effects (see section 3.7.4).

3.6.3 Testing in Incremental Development

Incremental development means that the project is not done in one (possibly large) piece but as a preplanned series of smaller developments and deliveries. System functionality and reliability will grow over time.

The objective of this is to make sure the system meets customer needs and expectations. The early releases allow customer feedback early and

15. This aspect is not shown in the general V-model. Only more modern life cycle models show iterations explicitly (see [Jacobson 99], [Beck 00], [Beedle 01]).

continuously. Examples of incremental models are Prototyping, Rapid Application Development (RAD) [Martin 91], Rational Unified Process (RUP), Evolutionary Development [Gilb 05], the Spiral Model [Boehm 86], and so-called agile development methods such as Extreme Programming (XP) [Beck 00], Dynamic Systems Development Method (DSDM) [Stapleton 02], and SCRUM [Beedle 01]. SCRUM has become more and more popular during recent years and is nowadays much used amongst agile approaches.

Testing must be adapted to such development models, and continuous integration testing and regression testing are necessary. There should be reusable test cases for every component and increment, and they should be reused and updated for every additional increment. If this is not the case, the product's reliability tends to decrease over time instead of increasing.

This danger can be reduced by running several V-models in sequence, one for each increment, where every next "V" reuses existing test material and adds the tests necessary for new development or for higher reliability requirements.

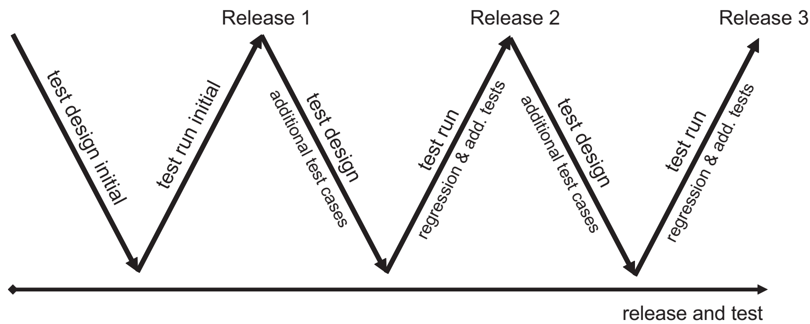


Figure 3–5
Testing in incremental development

3.7 Generic Types of Testing

The previous chapters gave a detailed view of testing in the software life cycle, distinguishing several test levels. Focus and objectives change when testing in these different levels. And different types of testing are relevant on each test level.

The following types of testing can be distinguished:

- Functional testing
- Nonfunctional testing
- Testing of software structure
- Testing related to changes

3.7.1 Functional Testing

Functional testing includes all kind of tests that verify a system's input/output behavior. To design functional test cases, the black box testing methods discussed in section 5.1 are used, and the test bases are the functional requirements.

Functional requirements

Functional requirements →specify the behavior of the system; they describe what the system must be able to do. Implementation of these requirements is a precondition for the system to be applicable at all. Characteristics of functionality, according to [ISO 9126], are suitability, accuracy, interoperability, and security.

Requirements definition

When a project is run using the V-model, the requirements are collected during the phase called "requirements definition" and documented in a requirements management system (see section 7.1.1). Text-based requirements specifications are still in use as well. Templates for this document are available in [IEEE 830].

The following text shows a part of the requirements paper concerning price calculation for the system VSR (see section 3.2.4).

Example: Requirements of the VSR-System

R 100: The user can choose a vehicle model from the current model list for configuration.
R 101: For a chosen model, the deliverable extra equipment items are indicated. The user can choose the desired individual equipment from this list.
R 102: The total price of the chosen configuration is continuously calculated from current price lists and displayed.

Requirements-based testing

Requirements-based testing uses the final requirements as the basis for testing. For each requirement, at least one test case is designed and documented in the test specification. The test specification is then reviewed. The testing of requirement 102 in the preceding example could look like the following example.

Example: Requirements-based testing

T 102.1: A vehicle model is chosen; its base price according to the sales manual is displayed.
T 102.2: A special equipment item is selected; the price of this accessory is added.
T 102.3: A special equipment item is deselected; the price falls accordingly.
T 102.4: Three special equipment items are selected; the discount comes into effect as defined in the specification.

Usually, more than one test case is needed to test a functional requirement.

Requirement 102 in the example contains several rules for different price calculations. These must be covered by a set of test cases (102.1–102.4 in the preceding example). Using black box test methods (e.g., →equivalence partitioning), these test cases can be further refined and extended if desired. The decisive fact is if the defined test cases (or a minimal subset of them) have run without failure, the appropriate functionality is considered validated.

Requirements-based functional testing as shown is mainly used in system testing and other higher levels of testing. If a software system's purpose is to automate or support a certain business process for the customer, business-process-based testing or use-case-based testing are other similar suitable testing methods (see section 5.1.5).

Example: Testing based on business process

From the dealer's point of view, VSR supports him in the sales process. The process can, for example, look like this:

- The customer selects a type of vehicle he is interested in from the available models.
- The customer gets the information about the type of extra equipment and prices and selects the desired car.
- The dealer suggests alternative ways of financing the car.
- The customer decides and signs the contract.

A business process analysis (which is usually elaborated as part of the requirements analysis) shows which business processes are relevant and how often and in which context they appear. It also shows which persons, enterprises, and external systems are involved. Test scenarios simulating typical business processes are constructed based on this analysis. The test

scenarios are prioritized using the frequency and the relevance of the particular business processes.

Requirements-based testing focuses on single system functions (e.g., the transmission of a purchase order). Business-process-based testing, however, focuses on the whole process consisting of many steps (e.g., the sales conversation, consisting of configuring a car, agreeing on the purchase contract, and the transmission of the purchase order). This means a sequence of several tests.

Of course, for the users of the *VirtualShowRoom* system, it is not enough to see if they can choose and then buy a car. More important for ultimate acceptance is often how easily they can use the system. This depends on how easy it is to work with the system, if it reacts quickly enough, and if it returns easily understood information. Therefore, along with the functional criteria, the nonfunctional criteria must also be checked and tested.

3.7.2 Nonfunctional Testing

→Nonfunctional requirements do not describe the functions; they describe the attributes of the functional behavior or the attributes of the system as a whole, i.e., "how well" or with what quality the (partial) system should work. Implementation of such requirements has a great influence on customer and user satisfaction and how much they enjoy using the product. Characteristics of these requirements are, according to [ISO 9126], reliability, usability, and efficiency. (For the new syllabus, which is effective from 2015, the basis is not ISO 9126 but ISO/IEC 25010:2011. Compatibility and security are added to the list of system characteristics.) Indirectly, the ability of the system to be changed and to be installed in new environments also has an influence on customer satisfaction. The faster and the easier a system can be adapted to changed requirements, the more satisfied the customer and the user will be. These two characteristics are also important for the supplier, because they help to reduce maintenance costs.

According to [Myers 79], the following nonfunctional system characteristics should be considered in the tests (usually in system testing):

- →**Load test:** Measuring of the system behavior for increasing system loads (e.g., the number of users that work simultaneously, number of transactions)
- →**Performance test:** Measuring the processing speed and response time for particular use cases, usually dependent on increasing load
- →**Volume test:** Observation of the system behavior dependent on the amount of data (e.g., processing of very large files)
- →**Stress test:** Observation of the system behavior when the system is overloaded
- **Testing of security** against unauthorized access to the system or data, denial of service attacks, etc.
- **Stability** or **reliability test:** Performed during permanent operation (e.g., mean time between failures or failure rate with a given user profile)
- →**Robustness test:** Measuring the system's response to operating errors, bad programming, hardware failure, etc. as well as examination of exception handling and recovery
- **Testing of compatibility and data conversion:** Examination of compatibility with existing systems, import/export of data, etc.
- **Testing of different configurations of the system:** For example, different versions of the operating system, user interface language, hardware platform, etc. (→back-to-back testing)
- **Usability test:** Examination of the ease of learning the system, ease and efficiency of operation, understandability of the system outputs, etc., always with respect to the needs of a specific group of users ([ISO 9241], [ISO 9126])
- **Checking of the documentation:** For compliance with system behavior (e.g., user manual and GUI)
- **Checking maintainability:** Assessing the understandability of the system documentation and whether it is up-to-date; checking if the system has a modular structure; etc.

A major problem in testing nonfunctional requirements is the often imprecise and incomplete expression of these requirements. Expressions like "the system should be easy to operate" and "the system should be fast" are not testable in this form.

Hint

- Representatives of the (later) system test personnel should participate in early requirement reviews and make sure that every nonfunctional requirement (as well as each functional one) can be measured and is testable.

Furthermore, many nonfunctional requirements are so fundamental that nobody really thinks about mentioning them in the requirements paper (presumed matters of fact).[16] Even such implicit characteristics must be validated because they may be relevant.

Example: Presumed requirements

The VSR-System is designed for use on a market-leading operating system. It is obvious that the recommended or usual user interface conventions are followed for the "look and feel" of the VSR GUI. The DreamCar GUI (see figure 3-3) violates these conventions in several aspects. Even if no particular requirement is specified, such deviations from "matter of fact requirements" can and must be seen as faults or defects.

Excursion: Testing nonfunctional requirements

In order to test nonfunctional characteristics, it makes sense to reuse existing functional tests. The nonfunctional tests are somehow "piggybacking" on the functional tests. Most nonfunctional tests are black box tests. An elegant general testing approach could look like this:

Scenarios that represent a cross section of the functionality of the entire system are selected from the functional tests. The nonfunctional property must be observable in the corresponding test scenario. When the test scenario is executed, the nonfunctional characteristic is measured. If the resulting value is inside a given limit, the test is considered "passed." The functional test practically serves as a vehicle for determining the nonfunctional system characteristics.

3.7.3 Testing of Software Structure

Structural techniques (→structure-based testing, white box testing) use information about the test object's internal code structure or architecture. Typically, the control flow in a component, the call hierarchy of procedures, or the menu structure is analyzed. Abstract models of the software may also be used. The objective is to design and run enough test cases to, if possible, completely cover all structural items. In order to do this, useful (and enough) test cases must be developed.

Structural techniques are most used in component and integration testing, but they can also be applied at higher levels of testing, typically as

16. This is regrettably also true for functional requirements. The "of course the system has to do X" implicit requirement is one of the main problems for testing.

extra tests (for example, to cover menu structures). Structural techniques are covered in detail in sections 4.2 and 5.2.

3.7.4 Testing Related to Changes and Regression Testing

When changes are implemented, parts of the existing software are changed or new modules are added. This happens when correcting faults and performing other maintenance activities. Tests must show that earlier faults are really repaired (➞retesting). Additionally, there is the risk of unwanted side effects. Repeating other tests in order to find them is called regression testing.

Regression test

A regression test is a new test of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made (uncovering masked defects).

Thus, regression testing may be performed at all test levels and applies to functional, nonfunctional, and ➞structural test. Test cases to be used in regression testing must be well documented and reusable. Therefore, they are strong candidates for ➞test automation.

The question is how extensive a regression test has to be. There are the following possibilities:

How much retest and regression test

1. Rerunning of all the tests that have detected failures whose reasons (the defects) have been fixed in the new software release (defect retest, confirmation testing)
2. Testing of all program parts that were changed or corrected (testing of altered functionality)
3. Testing of all program parts or elements that were newly integrated (testing of new functionality)[17]
4. Testing of the whole system (complete regression test)

A bare retest (1) as well as tests that execute only the area of modifications (2 and 3) are not enough because in software systems, simple local code changes can create side effects in any other, arbitrarily distant, system parts.

Changes can have unexpected side effects

If the test covers only altered or new code parts, it neglects the consequences these alterations can have on unaltered parts. The trouble with software is its complexity. With reasonable cost, it can only be roughly estimated where such unwanted consequences can occur. This is particu-

17. This is a regression test in a broader sense, where *changes* also means new functionality (see the glossary].

larly difficult for changes in systems with insufficient documentation or missing requirements, which, unfortunately, is often the case in old systems.

Full regression test

In addition to retesting the corrected faults and testing changed functions, all existing test cases should be repeated. Only in this case would the test be as safe as the testing done with the original program version. Such a complete regression test would also be necessary if the system environment has been changed because this could have an effect on every part of the system.

In practice, a complete regression test is usually too time consuming and expensive. Therefore, we are looking for criteria that can help to choose which old test cases can be omitted without losing too much information. As always, in testing this means balancing risk and cost. The following test selection strategies are often used:

Selection of regression test cases

- Repeating only the high-priority tests according to the test plan
- In the functional test, omitting certain variations (special cases)
- Restricting the tests to certain configurations only (e.g., testing of the English product version only, testing of only one operating system version)
- Restricting the test to certain subsystems or test levels

Excursion

Generally, the rules listed here refer to the system test. On the lower test levels, regression test criteria can also be based on design or architecture documents (e.g., class hierarchy) or white box information. Further information can be found in [Kung 95], [Rothermel 94], [Winter 98], and [Binder 99]. There, the authors not only describe special problems in regression testing object-oriented programs, they also describe the general principles of regression testing in detail.

3.8 Summary

- The general V-model defines basic test levels: component test, integration test, system test, and acceptance test. It distinguishes between verification and validation. These general characteristics of good testing are applicable to any life cycle model:
  - For every development step there is a corresponding test level
  - The objectives of testing are specific for each test level
  - The design of tests for a given test level should begin as early as possible, i.e., during the corresponding development activity

- Testers should be involved in reviewing development documents as early as possible
- The number and intensity of the test levels must be tailored to the specific needs of the project

- The V-model uses the fact that it is cheaper to repair defects a short time after they have been introduced. Thus, the V-model requires verification measures (for example, reviews) after ending every development phase. This way, the "ripple effect" of defects (i.e., more defects) is minimized.
- Component testing examines single software components. Integration testing examines the collaboration of these components. Functional and nonfunctional system testing examine the entire system from the perspective of the future users. In acceptance testing, the customer checks the product for acceptance respective to the contract and acceptance by users and operations personnel. If the system will be installed in many operational environments, then field tests provide an additional opportunity to get experience with the system by running preliminary versions.
- Defect correction (maintenance) and further development (enhancement) or incremental development continuously alter and extend the software product throughout its life cycle. All these altered versions must be tested again. A specific risk analysis should determine the amount of the regression tests.
- There are several types of test with different objectives: functional testing, nonfunctional testing, structure-based testing, and change-related testing.