

rockynook

> COMPUTING

4th Edition

Andreas Spillner, Tilo Linz, Hans Schaefer

# Software Testing Foundations

A Study Guide for the Certified Tester Exam

- Foundation Level
- ISTQB Compliant

## **Software Testing Foundations**

## About the Authors



**Andreas Spillner** is a professor of Computer Science in the Faculty of Electrical Engineering and Computer Science at Bremen University of Applied Sciences. For more than 10 years, he was president of the German Special Interest Group in Software Testing, Analysis, and Verification of the German Society for Informatics. He is a honorary member of the German Testing Board. His work emphasis is on software engineering, quality assurance, and testing.



**Tilo Linz** is CEO of imbus AG, a leading service company for software testing in Germany. He is president of the German Testing Board and was president of the ISTQB from 2002 to 2005. His work emphasis is on consulting and coaching projects on software quality management, and optimizing software development and testing processes.



**Hans Schaefer** is an independent consultant in software testing in Norway. He is president of the Norwegian Testing Board. He has been consulting and teaching software testing methods since 1984. He organizes the Norwegian Special Interest Group in Software Testing for Western Norway. His work emphasis is on consulting, teaching, and coaching test process improvement and test design techniques, as well as reviews.

Andreas Spillner · Tilo Linz · Hans Schaefer

# Software Testing Foundations

A Study Guide for the Certified Tester Exam

- Foundation Level
- ISTQB compliant

4th Edition

rockynook

Andreas Spillner (andreas.spillner@hs-bremen.de)  
Tilo Linz (tilo.linz@imbus.de)  
Hans Schaefer (hans.schaefer@ieee.org)

Editor: Dr. Michael Barabas  
Copyeditor: Judy Flynn  
Translator: Hans Schaefer  
Layout: Josef Hegele  
Project Manager: Matthias Rossmanith  
Cover Design: Helmut Kraus, www.exclam.de  
Printer: Sheridan  
Printed in the USA

ISBN 978-1-937538-42-2

4th Edition  
© 2014 by Spillner, Linz, Schaefer

Rocky Nook Inc.  
802 East Cota St., 3<sup>rd</sup> Floor  
Santa Barbara, CA 93103

www.rockynook.com

This 4<sup>th</sup> English book edition conforms to the 5<sup>th</sup> German edition “Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard” (dpunkt.verlag GmbH, ISBN: 978-3-86490-024-2), which was published in September 2012.

#### Library of Congress Cataloging-in-Publication Data

Spillner, Andreas.  
Software testing foundations / by Andreas Spillner, Tilo Linz, Hans Schaefer. -- Fourth edition.  
pages cm  
ISBN 978-1-937538-42-2 (paperback)  
1. Computer software--Testing. 2. Computer software--Verification. 3. Computer software--Evaluation. I. Linz, Tilo.  
II. Schaefer, H. (Hans) III. Title.  
QA76.76.T48S66 2014  
005.1'4--dc23

2013045349

Distributed by O'Reilly Media  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission of the publisher.

Many of the designations in this book used by manufacturers and sellers to distinguish their products are claimed as trademarks of their respective companies. Where those designations appear in this book, and Rocky Nook was aware of a trademark claim, the designations have been printed in caps or initial caps. All product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. They are not intended to convey endorsement or other affiliation with this book.

While reasonable care has been exercised in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein or from the use of the discs or programs that may accompany it.

This book is printed on acid-free paper.

---

# Preface

In most industrialized countries, the Certified Tester has gained acceptance as a training and education tool for testers. At the end of 2013, the number of certified testers worldwide was more than 300,000. Chris Carter, president of the International Software Testing Qualifications Board (ISTQB), says this: “I think the scheme has been so successful because we freely offer our syllabi and glossary to the public, which helps to standardize professional terminology. We also offer certifications at a range of levels, from foundation through advanced to expert, allowing testing professionals to be supported right through their careers and keeping them up-to-date with the world’s best practices.”

*Worldwide success*

There are more than 20,000 Certified Testers in Germany, more than 1,000 in Norway, and more than 2,000 in Sweden. Even the small country of Iceland has over 100 Certified Testers. In more and more countries, being a Certified Tester is a prerequisite to being employed in testing or to be a contractor in testing services.

*Certified Testers  
in some countries*

A 2011 poll (taken in Germany, Switzerland, and Austria) revealed that nearly 75% of the people asked know the ISTQB scheme. More than 70% of them already have a Foundation Level Certificate. About 90% said the training was helpful.

The first version of this book was published in German in 2002. The first English edition was published in 2006. The German issue is in its 5<sup>th</sup> edition and the English version is in its 4<sup>th</sup> edition. This book conforms to the ISTQB syllabus “Certified Software Tester—Foundation Level” version 2011. Most major changes planned for the 2015 version have been included and are specially marked.

*Ten-year anniversary of the  
German version of this book*

Ten years is a long time in the IT industry; new developments and paradigms are encouraged and used, and new and improved tools are available. On the other hand, there is some basic knowledge in computer science that does not change. In this book, we have concentrated on generic knowledge and techniques. We have not described techniques

whose benefits are yet unknown, or techniques that have to show their practical validity and applicability. The same is true about “special disciplines” in testing; testing of web applications, testing in agile projects, or testing of embedded or mobile systems, for example. These techniques are not part of the standard foundations. There is other literature about such specialized areas.

*Books for the  
advanced level*

The Certified Tester training scheme consists of three levels (see Chapter 1). Besides the foundation knowledge (Foundation Level) described in detail in this text, books are also available from Rocky Nook for the syllabus for the Advanced Level. These books are available:

- The Software Test Engineer’s Handbook [Bath 14] (for Test Analyst and Technical Test Analyst)
- Advanced Software Testing—Vol. 1 – 3 [Black 08, 09, 11]

Syllabi for the Expert Level also exist: “Improving the Test Process”<sup>1</sup> and “Test Management.” The syllabi for “Test Automation” and “Security Testing” are currently being finished.

*The knowledge is much  
asked for in the IT world*

The broad acceptance of this training scheme is made apparent by the powerful and continuous growth in ISTQB membership. 47 Testing Boards represent more than 70 countries. Ten years ago, there were a handful of members. Now ISTQB is represented in all parts of the world. The Certified Tester has grown to be a renowned trademark in the IT industry worldwide, and has considerably contributed to improving testing in the software development process.

*Testing is taught at colleges  
and universities*

The number of colleges that have integrated the Certified Tester scheme into their teaching is impressive. Courses are taught at places like Aachen and Bremen (Germany), Oslo (Norway), Reykjavik (Iceland), and Wismar (Germany). National Testing Boards usually decide which colleges offer these courses. Their relevance is shown by many job advertisements as well as requests for tenders. For personnel in software development it is more or less required to have some basic knowledge about testing, best shown by a certificate.

*Thank you*

We want to thank the colleagues from the German Testing Board and the ISTQB. Without their interest and work, the Certified Tester training scheme would not have received the success and acceptance described above.

---

1. [Bath 2013]

Why a new edition of this book? This edition contains corrections of faults and clarification of ambiguity, as far as we know them. A special thank you to the readers who have described faults and have asked us about the instances of ambiguity. Furthermore, the terminology has been made more consistent with the improved ISTQB-glossary. This edition of the book is consistent with the syllabus version 2011. The literature list was updated and new books and standards were included. The links to Internet pages were checked and updated. We wish all readers good luck when using the described testing approaches and techniques in practice and—when reading the book is part of the preparation for the Certified Tester examination—good luck with the exam.

*What has been changed*

Andreas Spillner and Tilo Linz  
Bremen, Möhrendorf, Germany  
August 2013

I want to especially thank Michael Barabas from dpunkt.verlag, the publisher of the German book, and Matthias Rossmanith from Rocky Nook for their support in preparing this book. There were a lot of late changes and delays, most of which can be attributed to me. My special thanks goes to Judy Flynn, copy editor at Rocky Nook. Without her help, this book would be much harder to read. She helped me to improve my English, without getting tired of my systematic errors. When translating the German book to English, I especially thought of readers who do not use English as their native language. Many of us use a different language in our life, but English for our business. I hope the book will be comprehensible to such readers.

I included some planned changes to the ISTQB syllabus. These are specially marked because they will not be included in exams before 2015. Most of them are obvious changes due to development in international standards. When taking the Certified Tester exam, please make sure you know which version of the syllabus is used in your exam!

Finally, the main goal for this book is that it should teach you how to test effectively and efficiently. You should learn that there is a lot more to learn in the area of testing. As a side effect, you should be prepared to pass the Certified Tester exam.

Hans Schaefer  
Valestrandsfossen, Norway  
February 2014



---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Fundamentals of Testing</b>	<b>5</b>
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
<b>3</b>	<b>Testing in the Software Life Cycle</b>	<b>39</b>
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

---

<b>4</b>	<b>Static Test</b>	<b>79</b>
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
<b>5</b>	<b>Dynamic Analysis – Test Design Techniques</b>	<b>105</b>
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
<b>6</b>	<b>Test Management</b> .....	<b>169</b>
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
<b>7</b>	<b>Test Tools</b>	<b>205</b>
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
<b>Appendix</b>		
<b>A</b>	<b>Test Plans According to IEEE Standard 829-1998</b>	<b>225</b>
	<b>Test Plans According to IEEE Standard 829-2008</b>	<b>231</b>
<b>B</b>	<b>Important Information about the Syllabus and the Certified Tester Exam</b>	<b>241</b>
<b>C</b>	<b>Exercises</b>	<b>243</b>
	<b>Glossary</b>	<b>247</b>
	<b>Literature</b>	<b>277</b>
	<b>Index</b>	<b>283</b>



---

# 1 Introduction

In recent years, software has 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.

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.

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.

*High dependence on the correct functioning of the software*

*Basic knowledge for structured evaluation and testing*

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.

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].

*Three-step qualification scheme*

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.

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.

*Foundations*

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.

*Testing in the software life cycle*

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.

*Static 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.

*Dynamic testing*

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.

---

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 (*JustIn-Time*). 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.

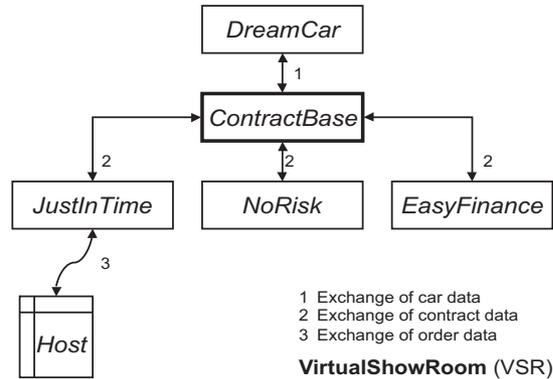
Figure 2-1 shows the general architecture of this software system.

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.

**Case study,**  
**"VirtualShowRoom" – VSR**

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.

---

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.

---

**Example**

### 2.1.1 Error, Defect, and Bug Terminology

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  $\rightarrow$ failure means that a given requirement is not fulfilled; it is a discrepancy between the  $\rightarrow$ actual result or behavior<sup>1</sup> and the  $\rightarrow$ expected result or behavior.<sup>2</sup>

*What is a defect, failure, or fault?*

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  $\rightarrow$ functional requirements.

In contrast to physical system failure, software failures do not occur because of aging or abrasion. They occur because of  $\rightarrow$ faults in the software. Faults (or defects or  $\rightarrow$ 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.

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  $\rightarrow$ tester or user; for example, an output is wrong or the program crashes.

*Failure*

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  $\rightarrow$ statements in the program.

*Fault*

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

*Defect masking*

- 
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  $\rightarrow$ error or  $\rightarrow$ 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  $\rightarrow$ debugging. Repairing a defect generally increases the  $\rightarrow$ quality of the product because the  $\rightarrow$ 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  $\rightarrow$ 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).

Every execution<sup>3</sup> (even using more or less random samples) of a  $\rightarrow$ test object in order to examine it is testing. The  $\rightarrow$ 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.<sup>4</sup>

*A test is a sample examination*

Testing software has different purposes:

- Executing a program to find failures
- Executing a program to measure quality
- Executing a program to provide confidence<sup>5</sup>
- 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).

Besides execution of the test object with  $\rightarrow$ test data, planning, design, implementation, and analysis of the test ( $\rightarrow$ test management) also belong to the  $\rightarrow$ test process. A  $\rightarrow$ test run or  $\rightarrow$ test suite includes execution of one or more  $\rightarrow$ 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].

*Testing terms*

Several test cases can often be combined to create  $\rightarrow$ 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.

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].

**Excursion:**  
**ISO/IEC 25010**

---

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.

---

**Example**  
**VirtualShowRoom**

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

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

*Functionality*

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 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).

*Efficiency*

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.

*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.

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.

*Prioritize quality characteristics*

#### 2.1.4 Test Effort

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.

*Complete testing is impossible*

---

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

**Example**

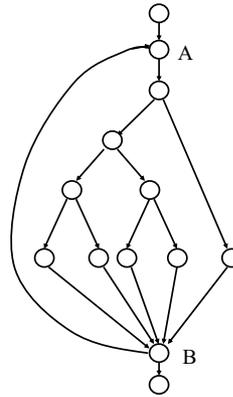
A small program with an easy control flow will be tested. The program consists of four decisions (IF-instructions) that are partially nested. The control flow graph of the program is shown in figure 2-2. Between Point A and B is a loop, with a return from Point B to Point A. If the program is supposed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

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

$5^1$  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.

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.

*Defects can cause high costs*

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<sup>6</sup> are lower than the costs of failure” [Koomen 99]. Thus, the test effort is always dependent on an estimation of the application risk.

---

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.

---

***Example for a high risk  
in case of failure***

---

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.

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.

*Limited resources*

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 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

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<sup>7</sup> [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].

**Excursion**  
**Life cycle models**

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"*

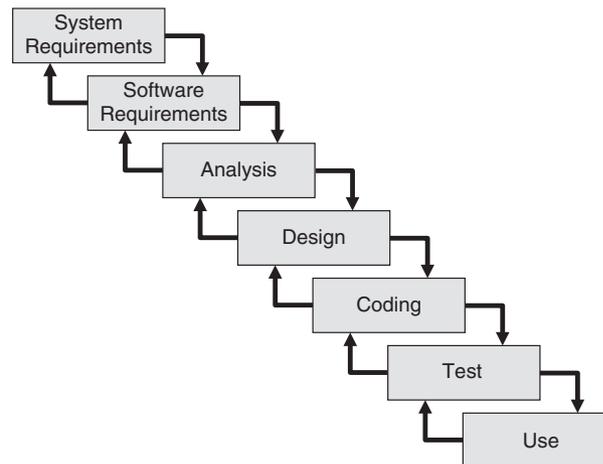
The first fundamental model was the waterfall model (see figure 2-3, shown with the originally defined phases [Royce 70]<sup>8</sup>). It is impressively simple and very well known. Only when one development phase is completed will the next one be initiated.

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*

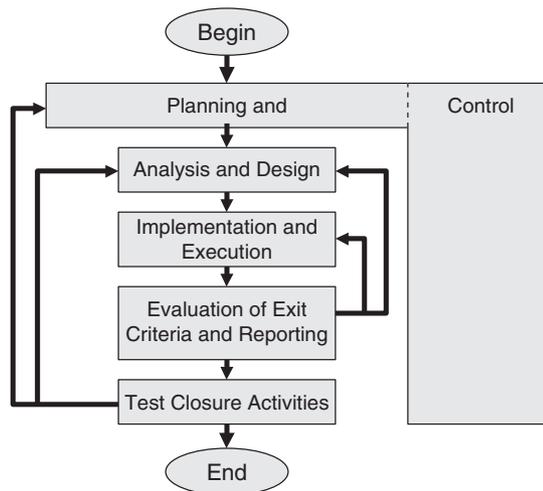
An enhancement of the waterfall model is the general V-model ([Boehm 79], [IEEE/IEC 12207]), where the constructive activities are decomposed from the testing activities (see chapter 3, figure 3-1). The model has the form of a V. The constructive activities, from requirements definition to implementation, 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.

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-

*Resource planning*

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.

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<sup>9</sup> are defined, they may be used after executing the test cases to decide if the test process can be finished.

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).

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

*Define test intensity for subsystems and different aspects*

*Prioritization of the tests*

*Tool support*

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.

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.

*Logical and concrete test cases*

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.

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 cases comprise more than just the test data*

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:

*Test oracle*

- 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<sup>10</sup>, 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)	$x < 15000$	$15000 \leq x \leq 20000$	$20000 < x < 25000$	$x \geq 25000$
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.

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

**Table 2-2***Table with concrete test cases*

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  $\rightarrow$ preconditions for the respective test case are fulfilled. The mutual traceability between test cases and specifications must be checked and, if necessary, updated.

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.

*Test case execution*

The test cases should also be grouped into  $\rightarrow$ 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).

Based on the  $\rightarrow$ 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.

*Correction may lead to new faults*

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  $\rightarrow$ risk-based testing (see section 6.4.3).

*The most important test cases first*

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<sup>11</sup>

*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

---

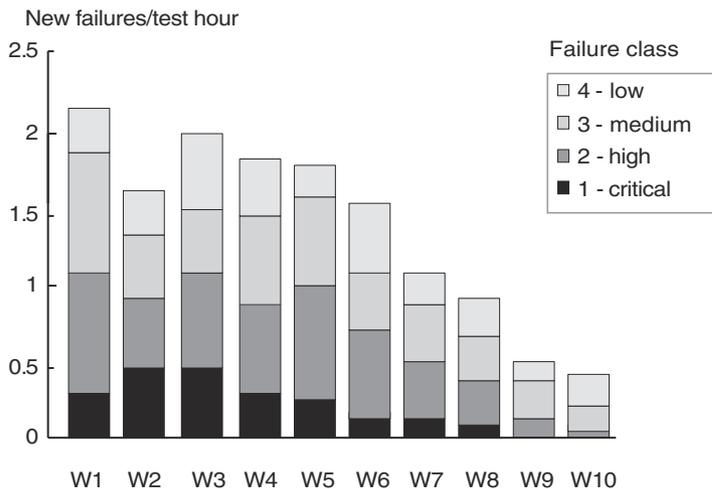
11. ISTQB calls this phase “Evaluation of test exit criteria and Reporting.”

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.

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.

*Further criteria for the determination of the test's end*



**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).

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

*Consider several test cycles*

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].

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.

*Archiving testware*

## 2.3 The Psychology of Testing

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.

*Errare humanum est*

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].

“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

*Developer test*

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 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.

*Mutual comprehension*

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 lose 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.

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<sup>12</sup> is as follows:

*Dealing with  
critical information*

---

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 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.

*The role of testing  
within life cycle models*

### 3.1 The General V-Model

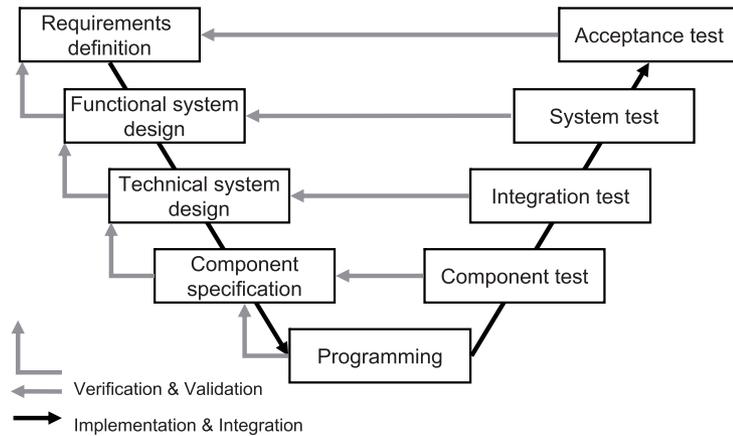
The main idea behind the general V-model is that development and testing tasks are corresponding activities of equal importance. The two branches of the *V* symbolize this.

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.<sup>1</sup>

---

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.

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

*Does a product solve the intended task?*

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

*Is it the right system?*

In addition to validation testing, the V-model requires verification<sup>3</sup> 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).

*Does a product fulfill its specification?*

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.

*Is the system correctly built?*

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<sup>4</sup> (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.

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.

*Test basis*

### 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.

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 test examines component internal aspects*

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.

---

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.

**Example:**  
**Testing of a class method**

- 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:<sup>5</sup>

```
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.

---

■ 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<sup>7</sup> 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.

---

**Hint**

---

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.

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):<sup>8</sup>

```
// 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);
```

**Example:**  
**Negative test**

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.

**Excursion**

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<sup>9</sup> and maintainability.

*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-

*Efficiency test*

- 
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.

---

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.<sup>10</sup> 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 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]).

**Example:**  
**Code as test basis**

*"Test first" development*

---

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*

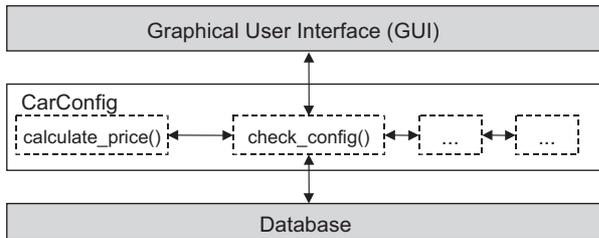
The test basis may be the software and system design or system architecture, or workflows through several interfaces and use cases.

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.

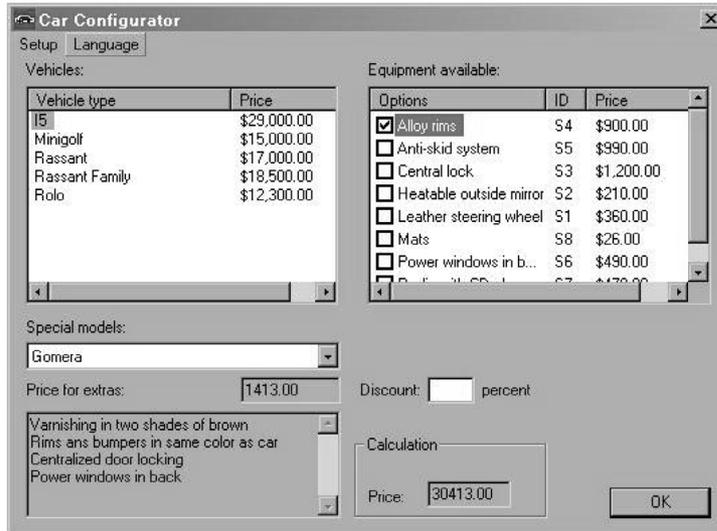
**Figure 3-2**  
*Structure of the subsystem*  
*VSR-DreamCar*



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.

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.

**Example:**  
**VSR integration test**

*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.

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.

*Reuse of the test environment*

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.

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.

*Monitors are necessary*

### 3.3.4 Test objectives

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

*Wrong interface formats*

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).

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:

*Typical faults in data exchange*

- 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.

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.

*Components are completed at different times*

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.

---

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  $\rightarrow$ 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

**Example:**  
**Integration Strategy**  
**in the VSR project**

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<sup>11</sup> might be chosen.

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:

*Avoid the big bang!*

- 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.<sup>12</sup> 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.

---

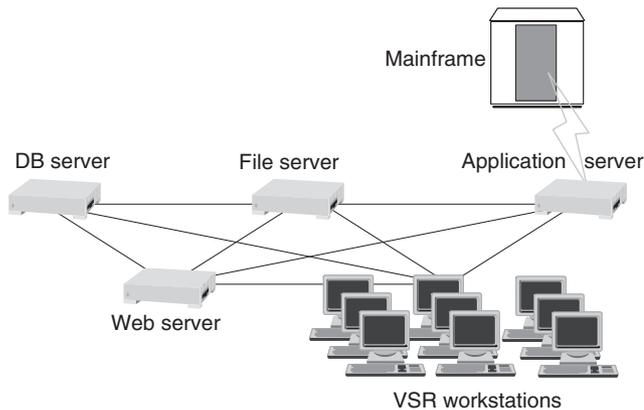
12. The customer (who has ordered and paid for the system) and the user (who uses the system) can be different groups of people or organizations with their own specific interests and requirements for the system.

### 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*

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.

→data quality

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:

*System test requires a separate test environment*

- 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]).

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.

*Missed decisions*

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.

*Project fail*

### 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.

In contrast to system testing, which takes place in the producer environment, acceptance testing is run in the customer's actual operational environment.<sup>13</sup> 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.

*Customer (site)  
acceptance test*

### 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.

---

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.

---

**Example:**  
**Different user groups**

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.

*Get acceptance of every  
user group*

---

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.<sup>14</sup> 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.

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.

*Dogfood test*

## 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.

---

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

**Example:**  
**Analysis of VSR hotline requests**

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.

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.

*Testing after  
maintenance work*

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 comprehensively 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.

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

*Testing before retirement*

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.

---

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.

**Example:**  
**Planning of the VSR  
development**

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.<sup>15</sup>

*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

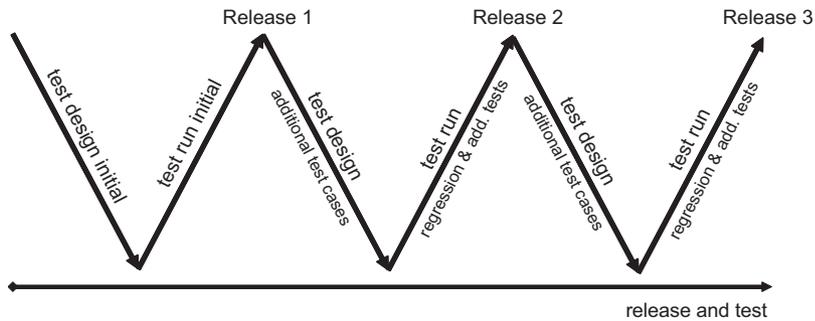
---

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.

- 
- 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.
- 

**Example:**  
**Requirements-based testing**

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).

---

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.
- 

**Example:**  
**Testing based on business process**

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).<sup>16</sup> 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.

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).

*Regression test*

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:

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)<sup>17</sup>
4. Testing of the whole system (complete regression test)

*How much retest and 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.

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-

*Changes can have unexpected side effects*

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.



---

## 4 Static Test

*Static investigations like reviews and tool-supported analysis of code and documents can be used very successfully for improving quality. This chapter presents the possibilities and techniques.*

An often-underestimated examination method is the so-called static test, often named static analysis. Opposite to dynamic testing (see chapter 5), the test object is not provided with test data and executed but rather analyzed. This can be done using one or more persons for an intensive investigation or through the use of tools. Such an investigation can be used for all documents relevant for software development and maintenance. Tool-supported static analysis is only possible for documents with a formal structure.

The goal of examination is to find defects and deviations from the existing specifications, standards to comply with, or even the project plan. An additional benefit of the results of these examinations is optimizing the development process. The basic idea is defect prevention: defects and deviations should be recognized as early as possible before they have any effect in the further development process where they would result in expensive rework.

### 4.1 Structured Group Evaluations

#### 4.1.1 Foundations

Reviews apply the human analytical capabilities to check and evaluate complex issues. Intensive reading and trying to understand the examined documents is the key.

There are different techniques for checking documents. They differ regarding the intensity, formality, necessary resources (staff and time), and goals.

*Systematic use of the human ability to think and analyze*

In the following sections, the different techniques are explained in more detail. Unfortunately, there is no uniform terminology concerning static analysis techniques. The terms used in this chapter are similar to the terms in the ISTQB syllabus and [IEEE 1028] (see the glossary in the appendix). Detailed descriptions can be found in [Freedman 90] and [Gilb 96].

### 4.1.2 Reviews

*Review* is a common generic term for all the different static analysis techniques people perform as well as the term for a specific document examination technique.

Another term, often used with the same meaning, is  $\rightarrow$ inspection. However, *inspection* is usually defined as a special, formal review using data collection and special rules [Fagan 76], [IEEE 1028], [Gilb 96]. All documents can be subjected to a review or an inspection, such as, for example, contracts, requirements definitions, design specifications, program code, test plans, and manuals. Often, reviews provide the only possibility to check the semantics of a document. Reviews rely on the colleagues of the author to provide mutual feedback. Because of this, they are also called peer reviews.

*A means for quality assurance*

Reviews are an efficient means to assure the quality of the examined documents. Ideally, they should be performed as soon as possible after a document is completed to find mistakes and inconsistencies early. The verifying examinations at the end of a phase in the general V-model normally use reviews (so-called phase exit reviews). Eliminating defects and inconsistencies leads to improved document quality and positively influences the whole development process because development is continued with documents that have fewer or even no defects.

*Positive effects*

In addition to defect reduction, reviews have further positive effects:

- Cheaper defect elimination. If defects are found and eliminated early, productivity in development increases because fewer resources are needed for defect identification and elimination later. These resources can instead be used for development.
- Shortened development time.
- If defects are recognized and corrected early, costs and time needed for executing the dynamic tests (see chapter 5) decrease because there are fewer defects in the test object.

- Because of the smaller number of defects, cost reduction can be expected during the whole product life. For example, a review may detect and clarify inconsistent and imprecise customer requests in the requirements. Foreseeable change requests after installation of the software system can thus be avoided.
- During operation of the system, a reduced failure rate can be expected.
- As the examinations are done using a team of people, reviews lead to mutual learning. People improve their working methods, and reviews will thus lead to enhanced quality of later products.
- Because several persons are involved in a review, a clear and understandable description of the facts is required. Often the necessity to formulate a clear document lets the author find forgotten issues.
- The whole team feels responsible for the quality of the examined object. The group will gain a common understanding of it.

The following problem can arise: In a badly moderated review session, the author may get into a psychologically difficult situation, feeling that he as a person and not the document is subject to critical scrutiny. Motivation to subject documents to a review will thus be destroyed. Concretely expressing the review objective, which is improving the document, may be helpful. One book [Freedman 90] extensively discusses how to solve problems with reviews.

*Potential problem*

The costs caused by reviews are estimated to be 10–15% of the development budget [Gilb 96, pg. 27]. The costs include the activities of the review process itself, analyzing the review results, and the effort put toward implementing them for process improvement. Savings are estimated to be about 14–25% [Bush 90]. The extra effort for the reviews themselves is included in this calculation.

*Reviews costs and savings*

If reviews are systematically used and efficiently run, more than 70% of the defects in a document can be found and repaired before they are unknowingly inherited by the next work steps [Gilb 96]. Considering that the costs for defect removal substantially increase in later development steps, it is plausible that defect cost in development is reduced by 75% and more.

- 
- Documents with a formal structure should be analyzed using a (static analysis) tool that checks this structure before the review. The tool can examine many aspects and can detect defects or deviations that do not need to be checked in a review (see section 4.2)
- 

**Hint**

*Important success factors* The following factors are decisive for success when using reviews (as suggested by [IEEE 1028]):

- Every review has a clear goal, which is formulated beforehand.
- The “right” people are chosen as review participants based on the review objective as well as on their subject knowledge and skills.

### 4.1.3 The General Process

The term *review* describes a whole group of static examinations. Section 4.1.5 describes the different types of reviews. The process underlying all reviews is briefly described here in accordance with the IEEE Standard for Software Reviews [IEEE 1028].

A review requires six work steps: planning, kick-off, individual preparation, review meeting, rework, and follow-up.

#### Planning

*Reviews need planning* Early, during overall planning, management must decide which documents in the software development process shall be subject to which review technique. The estimated effort must be included in the project plans. Several analyses show optimal checking time for reviewing documents and code [Gilb 96]. During planning of the individual review, the review leader selects technically competent staff and assembles a review team. In cooperation with the author of the document to be reviewed, she makes sure that the document is in a reviewable state, i.e., it is complete enough and reasonably finished. In formal reviews, entry criteria (and the corresponding exit criteria) may be set. A review should continue only after any available entry criteria has been checked.

*Different perspectives increase the effect* A review is, in most cases, more successful when the examined document is read from different viewpoints or when each person checks only particular aspects. The viewpoints or aspects to be used should be determined during review planning. A review might not involve the whole document. Parts of the document in which defects constitute a high risk could be selected. A document may also be sampled only to make a conclusion about the general quality of the document.

If a kick-off meeting is necessary, the place and time must be agreed upon.

### **Kick-Off**

The kick-off (or overview) serves to provide those involved in the review with all of the necessary information. This can happen through a written invitation or a meeting when the review team is organized. The purpose is sharing information about the document to be reviewed (*the review object*) and the significance and the objective of the planned review. If the people involved are not familiar with the domain or application area of the review object, then a short introduction to the material may be arranged, and a description of how it fits into the application or environment may be provided.

In addition to the review object, those involved must have access to other documents. These include the documents that help to decide if a particular statement is wrong or correct. The review is done against these documents (e.g., requirements specification, design, guidelines, or standards). Such documents are also called base documents or baselines. Furthermore, review criteria (for example, checklists) are very useful for supporting a structured process.

*Higher-level documents  
are necessary*

For more formal reviews, the entry criteria might be checked. If entry criteria are not met, the review should be canceled, saving the organization time that would otherwise be wasted reviewing material that may be “immature,” i.e., not good enough.

### **Individual Preparation**

The members of the review team must prepare individually for the review meeting. A successful review meeting is only possible with adequate preparation.

*Intensive study  
of the review object*

The reviewers intensively study the review object and check it against the documents given as a basis for it as well as against their own experience. They note deficiencies (even any potential defects), questions, or comments.

### **Review Meeting**

A review leader or →moderator leads the review meeting. Moderator and participants should behave diplomatically (not be aggressive with each other) and contribute to the review in the best possible way.

The review leader must ensure that all experts will be able to express their opinion knowing that the product will be evaluated and not the

author. Conflicts should be prevented. If this is not possible, a solution for the situation should be found.

Usually, the review meeting has a fixed time limit. The objective is to decide if the review object has met the requirements and complies with the standards and to find defects. The result is a recommendation to accept, repair, or rewrite the document. All the reviewers should agree upon the findings and the overall result.

*Rules for review meetings* Here are some general rules for a review meeting:<sup>1</sup>

1. The review meeting is limited to two hours. If necessary, another meeting is called, but it should not take place on the same day.
2. The moderator has the right to cancel or stop a meeting if one or more experts (reviewers) don't appear or if they are not sufficiently prepared.
3. The document (the review object) is subject to discussion, not the author:
  - The reviewers have to watch their expressions and their way of expressing themselves.
  - The author should not defend himself or the document. (That means, the author should not be attacked or forced into a defensive position. However, justification or explanation of the author's decisions is often seen as legitimate and helpful.)
4. The moderator should not also be a reviewer at the same time.
5. General style questions (outside the guidelines) shall not be discussed.
6. Solutions and discussing them isn't a task of the review team.
7. Every reviewer must have the opportunity to adequately present his or her issues.
8. The protocol must describe the consensus of the reviewers.
9. Issues must not be written as commands to the author (additional concrete suggestions for improvement or correction are sometimes considered useful and sensible for quality improvement).
10. The issues should be weighted<sup>2</sup> as follows:
  - Critical defect (the review object is not suitable for its purpose, the defect must be corrected before the object is approved)

---

1. Some of these rules only apply for some of the review techniques described in [IEEE 1028].

2. See section 6.6.3: Defects of severity class 2 and 3 can be seen as major defects and class 4 and 5 as minor defects.

- Major defect (the usability of the review object is affected, the defect must be corrected before the approval)
  - Minor defect (small deviation, for example, spelling error or bad expression, hardly affects the usage)
  - Good (flawless, this area should not be changed during rework).
11. The review team shall make a recommendation about the acceptance of the review object (see follow-up):
- Accept (without changes)
  - Accept (with changes, no further review)
  - Do not accept (further review or other checking measures are necessary)
12. Finally, all the session participants should sign the protocol

The protocol contains a list of the issues/findings that were discussed during the meeting. An additional review summary report should collect all important data about the review itself, i.e., the review object, the people involved, their roles (see section 4.1.4), a short summary of the most important issues, and the result of the review with the recommendation of the reviewers. In a more formal review, the fulfillment of formal exit criteria may be documented. If there was no physical meeting and, for example, electronic communication was used instead, there should definitely be a protocol.

*Protocol and summary  
of results*

### **Rework**

The manager decides whether to follow the recommendation or do something else. A different decision is, however, the sole responsibility of the manager. Usually, the author will eliminate the defects on the basis of the review results and rework the document. More formal reviews additionally require updating the defect status of every single found defect.

### **Follow-Up**

The proper correction of defects must be followed up, usually by the manager, moderator, or someone especially assigned this responsibility.

If the result of the first review was not acceptable, another review should be scheduled. The process described here can be rerun, but usually it is done in an abbreviated manner, checking only changed areas.

*Second review*

The review meetings and their results should then be thoroughly evaluated to improve the review process, to adapt the used guidelines and

*Find and fix deficiencies in  
the software development  
process*

checklists to the specific conditions, and to keep them up-to-date. To achieve this, it is necessary to collect and evaluate measurement data.

Recurring, or frequently occurring, defect types point to deficiencies in the software development process or lack of technical knowledge of the people involved. Necessary improvements of the development process should be planned and implemented. Such defect types should be included in the checklists. Training must compensate for lack of technical knowledge.

For more formal reviews, the final activity is checking the exit criteria. If they are met, the review is finished. Otherwise, it must be determined whether rework can be done or if the whole review was unsuccessful.

#### **4.1.4 Roles and Responsibilities**

The description of the general approach included some information on roles and responsibilities. This section presents the people involved and their tasks.

*Manager*

The manager selects the objects to be reviewed, assigns the necessary resources, and selects the review team.

Representatives of the management level should not participate in review meetings because management might evaluate the qualifications of the author and not the document. This would inhibit a free discussion among the review participants. Another reason is that the manager often lacks the necessary detailed understanding of technical documents. In a review, the technical content is checked, and thus the manager would not be able to add valuable comments. Management reviews of project plans and the like are a different thing. In this case, knowledge of management principles is necessary.

*Moderator*

The moderator is responsible for executing the review. Planning, preparation, execution, rework, and follow-up should be done in such a way that the review objectives are achieved.

The moderator is responsible for collecting review data and issuing the review report.

This role is crucial for the success of the review. First and foremost, a moderator must be a good meeting leader, leading the meeting efficiently and in a diplomatic way. A moderator must be able to stop unnecessary discussions without offending the participants, to mediate when there are conflicting points of view, and be able to see “between the lines.” A moderator must be neutral and must not state his own opinion about the review object.

The author is the creator of the document that is the subject of a review. If several people have been involved in the creation, one person should be appointed to be responsible; this person assumes the role of the author. The author is responsible for the review object meeting its review entry criteria (i.e., that the document is reviewable) and for performing any rework required for meeting the review exit criteria.

*Author*

It is important that the author does not interpret the issues raised on the document as personal criticism. The author must understand that a review is done only to help improve the quality of the product.

The reviewers, sometimes also called inspectors, are several (usually a maximum of five) technical experts that participate in the review meeting after necessary individual preparation.

*Reviewer*

They identify and describe problems in the review object. They should represent different viewpoints (for example, sponsor, requirements, design, code, safety, test). Only those viewpoints pertinent to the review of the product should be considered.

Some reviewers should be assigned specific review topics to ensure effective coverage. For example, one reviewer might focus on conformance with a specific standard, another on syntax. The manager should assign these roles when planning the review.

The reviewers should also label the good parts in the document. Insufficient or deficient parts of the review object must be labeled accordingly, and the deficiencies must be documented for the author in such a way that they can be corrected.

The recorder (or scribe) shall document the issues (problems, action items, decisions, and recommendations) found by the review team.

*Recorder*

The recorder must be able to record in a short and precise way, correctly capturing the essence of the discussion. This may not be easy because contributions are often not clearly or well expressed. Pragmatic reasons may make it meaningful to let the author be recorder. The author knows exactly how precisely and how detailed the contributions of the reviewers need to be recorded in order to have enough information for rework.

### **Possible difficulties**

Reviews may fail to achieve their objectives due to several causes:

*Reasons for less successful reviews*

- The required persons are not available or do not have the required qualifications or technical skills. This may be solved by training or by

using qualified staff from consulting companies. This is especially true for the moderator, because he must have more psychological than technical skills.

- Inaccurate estimates during resource planning by management may result in time pressure, which then causes unsatisfactory review results. Sometimes, a less costly review type can bring relief.
- If reviews fail due to lack of preparation, this is mostly because the wrong reviewers were chosen. If a reviewer does not realize the importance of the review and its great effect on quality improvement, then figures must be shown that prove the productive benefit of reviews. Other reasons for review failure may be lack of time and lack of motivation.
- A review can also fail because of missing or insufficient documentation. Prior to the review, it must be verified that all the needed documents exist and that they are sufficiently detailed. Only when this is the case should a review be performed.
- The review process cannot be successful if there is lack of management support because the necessary resources will not be provided and the results will not be used for process improvement. Unfortunately, this is often the case.

Detailed advice for solving these problems is described in [Freedman 90].

#### 4.1.5 Types of Reviews

Two main groups of reviews can be distinguished depending on the review object to be examined:

- Reviews pertaining to products or intermediate products that have been created during the development process
- Reviews that analyze the project itself or the development process

##### **Excursion**

Reviews in the second group are called →management reviews<sup>3</sup> [IEEE 1028] or project reviews. Their objective is to analyze the project itself or the development process. For example, such a review determines if plans and rules are followed, if the necessary work tasks are executed, or the effectiveness of process improvements or changes.

---

3. In [ISO 9000] the management review is defined in a more narrow way as “a formal evaluation by top management of the status and adequacy of the quality system in relation to the quality policy and objectives.”

The project as a whole and determining its current state are the objects of such a review. The state of the project is evaluated with respect to technical, economic, time, and management aspects.

Management reviews are often performed when reaching a milestone in the project, when completing a main phase in the software development process, or as a “postmortem” analysis to learn from the finished project.

In the following sections, the first group of reviews is described in more detail. We can distinguish between the following review types: →walkthrough, inspection, →technical review, and →informal review. In the descriptions, the focus is on the main differences between the particular review type and the basic review process (see section 4.1.3).

### Walkthrough

A walkthrough<sup>4</sup> is a manual, informal review method with the purpose of finding defects, ambiguities, and problems in written documents. The author presents the document to the reviewers in a review meeting.

Educating an audience regarding a software product is mentioned in [IEEE 1028] as a further purpose of walkthroughs. Further objectives of walkthroughs are to improve the product, to discuss alternative implementations, and to evaluate conformance to standards and specifications.

The main emphasis of a walkthrough is the review meeting (without a time limit). There is less focus on preparation compared to the other types of reviews; it can even be omitted sometimes.<sup>5</sup>

In most cases, typical usage situations, also called scenarios, will be discussed. Test cases may even be “played through.” The reviewers try to find possible errors and defects by spontaneously asking questions.

The technique is useful for small teams of up to five persons. It does not require a lot of resources because preparation and follow-up are minor or sometimes not even required. The walkthrough is useful for checking “noncritical” documents.

The author chairs the meeting and therefore has a great amount of influence. This can have a detrimental effect on the result if the author impedes an intensive discussion of the critical parts of the review object.

The author is responsible for follow-up; there is no more checking.

*Discussion of typical usage situations*

*Suitable for small development teams*

4. Also called “structured walkthrough”

5. According to [IEEE 1028], the participants should receive the documents in advance and should have prepared for the meeting.

Before the meeting the reviewers prepare, the results are written in a protocol, and someone other than the author records the findings. In practice there is a wide variation from informal to formal walkthroughs.

*Objectives*

The main objectives of a walkthrough are mutual learning, development of an understanding of the review object, and error detection.

**Inspection**

*Formal process*

The inspection is the most formal review. It follows a formal, prescribed process. Every person involved, usually people who work directly with the author, has a defined role. Rules define the process. The reviewers use checklists containing criteria for checking the different aspects.

The goals are finding unclear items and possible defects, measuring review object quality, and improving the quality of the inspection process and the development process. The concrete objectives of each individual inspection are determined during planning. The inspectors (reviewers) prepare for only a specific number of aspects that will be examined. Before the inspection begins, the inspection object is formally checked with respect to entry criteria and reviewability. The inspectors prepare themselves using procedures or standards and checklists.

Traditionally, this method of reviewing has been called design inspection or code or software inspection. The name points to the documents that are subject to the inspection (see [Fagan 76]). However, inspections can be used for any document in which formal evaluation criteria exist.

*Inspection meeting*

A moderator leads the meeting. The inspection meeting follows this agenda:

- The moderator first presents the participants and their roles as well as a short introduction to the topic of the inspection object.
- The moderator asks the participants if they are adequately prepared. In addition, the moderator might ask how much time the reviewer used to prepare and how many and how severe were the issues found.
- The group may review the checklists chosen for the inspection in order to make sure everyone is well prepared for the meeting.
- Issues of a general nature concerning the whole inspection object are discussed first and written into the protocol.

- A reviewer presents<sup>6</sup> the contents of the inspection object quickly and logically. If it's considered useful, passages can also be read aloud.
- The reviewers ask questions during this procedure, and the selected aspects of the inspection are thoroughly discussed. The author answers questions. The moderator makes sure that a list of issues is written. If author and reviewer disagree about an issue, a decision is made at the end of the meeting.
- The moderator must intervene if the discussion is getting out of control. The moderator also makes sure the meeting covers all aspects to be evaluated as well as the whole document. The moderator makes sure the recorder writes down all the issues and ambiguities that are detected.
- At the end of the meeting, all recorded items are reviewed for completeness.
- Discussions are conducted to resolve disagreements, for example, whether or not something can be classified a defect. If no resolution is reached, this is written in the protocol. There should be no discussion on how to solve the issues. Any discussion should be limited in time.
- Finally, the reviewers judge the inspection object as a whole.
- They decide if it must be reworked or not. In inspections, follow-up and reinspection are formally regulated.

In an inspection, data are also collected for general quality assessment of the development process and the inspection process. Therefore, an inspection also serves to optimize the development process, in addition to assessing the inspected documents. The collected data are analyzed in order to find causes for weaknesses in the development process. After process improvement, comparing the collected data before the change to the current data checks the improvement effect.

*Additional assessment of the development and inspection process*

The main objective of inspection is defect detection or, more precisely, the detection of defects causes and defects.

*Objective*

### Technical Review

In a technical review, the focus is compliance of the document with the specification, fitness for its intended purpose, and compliance to standards.

*Does the review object fulfill its purpose?*

---

6. Often, reviewers are called *inspectors*. [IEEE 1028] calls the presenting reviewer the *reader*.

During preparation, the reviewers check the review object with respect to the specified review criteria.

*Technical experts as reviewers*

The reviewers must be technically qualified. Some of them should not be project participants in order to avoid “project blindness.” Management does not participate. Basis for the review is only the “official” specification and the specified criteria for the review. The reviewers write down their comments and pass them to the moderator before the review meeting.<sup>7</sup> The moderator (who ideally is properly trained) sorts these findings based on their presumed importance. During the review meeting, only selected remarks are discussed.

*High preparation effort*

Most of the effort is in preparation. The author does not normally attend the meeting. During the meeting, the recorder notes all the issues and prepares the final documentation of the results.

The review result must be approved unanimously by all involved reviewers and signed by everyone. Disagreement should be noted in the protocol. It is not the job of the review participants to decide on the consequences of the result; that is the responsibility of management. If the review is highly formalized, entry and exit criteria of the individual review steps may also be defined.

In practice, very different versions of the technical review are found, from a very informal to a strictly defined, formal process.

*Objective*

Discussion is expressly requested during a technical review. Alternative approaches should be considered and decisions made. The specialists may solve the technical issues. The conformity of the review object with its specifications and applicable standards can be assessed. Technical reviews can, of course, reveal errors and defects.

### **Informal Review**

The informal review is a light version of a review. However, it more or less follows the general procedure for reviews (see section 4.1.3) in a simplified way. In most cases, the author initiates an informal review. Planning is restricted to choosing reviewers and asking them to deliver their remarks at a certain point in time. Often, there is no meeting or exchange of the findings. In such cases, the review is just a simple author-reader-cycle. The informal review is a kind of cross reading by one or more colleagues. The results need not be explicitly documented; a list of remarks or the revised

---

7. In [IEEE 1028], this also applies to inspection.

document is in most cases enough. Pair programming, buddy testing, code swapping, and the like are types of informal review. The informal review is very common and highly accepted due to the minimal effort required.

An informal review involves relatively little effort and low costs. Discussion and exchange of information among colleagues are welcome “side effects” of the process.

*Objective*

### **Selection Criteria**

The type of review that should be used depends very much on how thorough the review needs to be and the effort that can be spent. It also depends on the project environment; we cannot give specific recommendations. The decision about what type of review is appropriate must be made on a case-by-case basis. Here are some questions and criteria that should help:

*Selecting the type of review*

- The form in which the results of the review should be presented can help select the review type. Is detailed documentation necessary, or is it good enough to present the results informally?
- Will it be difficult or easy to find a date and time for the review? It can be difficult to bring together five or seven technical experts for one or more meetings.
- Is it necessary to have technical knowledge from different disciplines?
- What level (how deep) of technical knowledge is required for the review object? How much time will the reviewers need?
- Is the preparation effort appropriate with respect to the benefit of the review (the expected result)?
- How formally written is the review object? Is it possible to perform tool-supported analyses?
- How much management support is available? Will management curtail reviews when the work is done under time pressure?

It makes sense to use testers as reviewers. The reviewed documents are usually used as the test basis to design test cases. Testers know the documents early and they can design test cases early. By looking at documents from a testing point of view, testers may check new quality aspects, such as testability.

*Testers as reviewers*

### Notes

As we said in the beginning of the chapter, there are no uniform descriptions of the individual types of review. There is no clear boundary between the different review types, and the same terms are used with different meanings.

#### *Company-specific reviews*

Generally, it can be said that the types of reviews are very much determined by the organization that uses them. Reviews are tailored to the specific needs and requirements of a project. This has a positive influence on their efficiency.

A cooperative collaboration between the people involved in software development can be considered beneficial to quality. If people examine each other's work results, defects and ambiguities can be revealed. From this point of view, pair programming, as suggested in →Extreme Programming, can be regarded as a permanent "two-person-review" [Beck 00].

With distributed project teams, it might be hard to organize review meetings. These days, reviews can be in the form of structured discussion by Internet, videoconferencing, telephone conference calls, etc.

### Success Factors

The following factors are crucial for review success and must be considered:

- Reviews help improve the examined documents. Detecting issues, such as unclear points and deviations, is a wanted and required effect. The issues must be formulated in a neutral and objective way.
- Human and psychological factors have a strong influence in a review. A review must be conducted in an atmosphere of trust. The participants must be sure that the outcome will not be used to evaluate them (for example, as a basis of their next job assessment). It's important that the author of the reviewed document has a positive experience.
- Testers should be used as reviewers. They contribute to the review by finding (testing) issues. When they participate in reviews, testers learn about the product, which enables them to prepare tests earlier and in a better way.
- The type and level of the examined document, and the state of knowledge of the participating people, should be considered when choosing the type of review to use.
- Checklists and guidelines should be used to help in detecting issues during reviews.

- Training is necessary, especially for more formal types of reviews, such as inspections.
- Management can support a good review process by allocating enough resources (time and personnel) for document reviews in the software development process.
- Continuous learning from executed reviews improves the review process and thus is important.

## 4.2 Static Analysis

The objective of static analysis is, as with reviews, to reveal defects or defect-prone parts in a document. However, in static analysis, tools do the analysis. For example, even spell checkers can be regarded as a form of →static analyzers because they find mistakes in documents and therefore contribute to quality improvement.

*Analysis without executing the program*

The term *static analysis* points to the fact that this form of checking does not involve an execution of the checked objects (of a program). An additional objective is to derive measurements, or metrics, in order to measure and prove the quality of the object.

The document to be analyzed must follow a certain formal structure in order to be checked by a tool. Static analysis makes sense only with the support of tools. Formal documents can note, for example, the technical requirements, the software architecture, or the software design. An example is the modeling of class diagrams in UML.<sup>8</sup> Generated outputs in HTML<sup>9</sup> or XML<sup>10</sup> can also be subjected to tool-supported static analysis. Formal models developed during the design phases can also be analyzed and inconsistencies can be detected. Unfortunately, in practice, the program code is often the one and only formal document in software development that can be subjected to static analysis.

*Formal documents*

Developers typically use static analysis tools before or during component or integration testing to check if guidelines or programming conventions are adhered to. During integration testing, adherence to interface guidelines is analyzed.

Analysis tools often produce a long list of warnings and comments. In order to effectively and efficiently use the tools, the mass of generated

---

8. Unified Modeling Language [URL: UML]

9. HyperText Markup Language [URL: HTML]

10. Extensible Markup Language [URL: XML]

information must be handled intelligently; for example, by configuring the tool. Otherwise, the tools might be avoided.

*Static analysis and reviews*

Static analysis and reviews are closely related. If a static analysis is performed before the review, a number of defects and deviations can be found and the number of the aspects to be checked in the review clearly decreases. Due to the fact that static analysis is tool supported, there is much less effort involved than in a review.

**Hint**

- 
- If documents are formal enough to allow tool-supported static analysis, then it should definitely be performed before the document reviews because faults and deviations can be detected conveniently and cheaply and the reviews can be shortened.
  - Generally, static analysis should be used even if no review is planned. Each time a discrepancy is located and removed, the quality of the document increases.
- 

Not all defects can be found using static testing, though. Some defects become apparent only when the program is executed (that means at runtime) and cannot be recognized before. For example, if the value of the denominator in a division is stored in a variable, that variable can be assigned the value zero. This leads to a failure at runtime. In static analysis, this defect cannot easily be found, except for when the variable is assigned the value zero by a constant having zero as its value.

All possible paths through the operations may be analyzed, and the operation can be flagged as potentially dangerous. On the other hand, some inconsistencies and defect-prone areas in a program are difficult to find by dynamic testing. Detecting violation of programming standards or use of forbidden error-prone program constructs is possible only with static analysis (or reviews).

*The compiler is an analysis tool*

All compilers carry out a static analysis of the program text by checking that the correct syntax of the programming language is used. Most compilers provide additional information, which can be derived by static analysis (see section 4.2.1). In addition to compilers, there are other tools that are so-called analyzers. These are used for performing special analyses or groups of analyses.

The following defects and dangerous constructions can be detected by static analysis:

- Syntax violations
- Deviations from conventions and standards

- →Control flow anomalies
- →Data flow anomalies

Static analysis can be used to detect security problems. Many security holes occur because certain error-prone program constructs are used or necessary checks are not done. Examples are lack of buffer overflow protection and failing to check that input data may be out of bounds. Tools can find such deficiencies because they often search and analyze certain patterns.

*Finding security problems*

### 4.2.1 The Compiler as a Static Analysis Tool

Violation of the programming language syntax is detected by static analysis and reported as a fault or warning. Many compilers also generate further information and perform other checks:

- Generating a cross-reference list of the different program elements (e.g., variables, functions)
- Checking for correct data type usage by data and variables in programming languages with strict typing
- Detecting undeclared variables
- Detecting code that is not reachable (so-called →dead code)
- Detecting overflow or underflow of field boundaries (with static addressing)
- Checking interface consistency
- Detecting the use of all labels as jump start or jump target

The information is usually provided in the form of lists. A result reported as “suspicious” by the tool is not always a fault. Therefore, further investigation is necessary.

### 4.2.2 Examination of Compliance to Conventions and Standards

Compliance to conventions and standards can also be checked with tools. For example, tools can be used to check if a program follows programming regulations and standards. This way of checking takes little time and almost no personnel resources. In any case, only guidelines that can be verified by tools should be accepted in a project. Other regulations usually prove to be bureaucratic waste anyway. Furthermore, there often is an additional advantage: if the programmers know that the program code is checked for compliance to the programming guidelines, their willingness

to work according to the guidelines is much higher than without an automatic test.

### 4.2.3 Execution of Data Flow Analysis

*Checking the use of data*

→Data flow analysis is another means to reveal defects. Here, the usage of data on →paths through the program code is checked. It is not always possible to decide if an issue is a defect. Instead, we speak of →anomalies, or data flow anomalies. An anomaly is an inconsistency that can lead to failure but does not necessarily do so. An anomaly may be flagged as a risk.

An example of a data flow anomaly is code that reads (uses) variables without previous initialization or code that doesn't use the value of a variable at all. The analysis checks the usage of every single variable. The following three types of usage or states of variables are distinguished:

- **Defined (d)**: The variable is assigned a value.
- **Referenced (r)**: The value of the variable is read and/or used.
- **Undefined (u)**: The variable has no defined value.

*Data flow anomalies*

We can distinguish three types of data flow anomalies:

- **ur-anomaly**: An undefined value (u) of a variable is read on a program path (r).
- **du-anomaly**: The variable is assigned a value (d) that becomes invalid/undefined (u) without having been used in the meantime.
- **dd-anomaly**: The variable receives a value for the second time (d) and the first value had not been used (d).

**Example of anomalies**

We will use the following example (in C++) to explain the different anomalies. The following function is supposed to exchange the integer values of the parameters Max and Min with the help of the variable Help, if the value of the variable Min is greater than the value of the variable Max:

```
void exchange (int& Min, int& Max) {
    int Help;
    if (Min > Max) {
        Max = Help;
        Max = Min;
        Help = Min;
    }
}
```

After the usage of the single variables is analyzed, the following anomalies can be detected:

- **ur-anomaly** of the variable `Help`: The domain of the variable is limited to the function. The first usage of the variable is on the right side of an assignment. At this time, the variable still has an undefined value, which is referenced there. There was no initialization of the variable when it was declared (this anomaly is also recognized by usual compilers, if a high warning level is activated).
- **dd-anomaly** of the variable `Max`: The variable is used twice consecutively on the left side of an assignment and therefore is assigned a value twice. Either the first assignment can be omitted or the programmer forgot that the first value (before the second assignment) has been used.
- **du-anomaly** of the variable `Help`: In the last assignment of the function, the variable `Help` is assigned another value that cannot be used anywhere because the variable is valid only inside the function.

In this example, the anomalies are obvious. But it must be considered that between the particular statements that cause these anomalies there could be an arbitrary number of other statements. The anomalies would not be as obvious anymore and could easily be missed by a manual check such as, for example, a review. A tool for analyzing data flow can, however, detect the anomalies.

*Data flow anomalies are usually not that obvious*

Not every anomaly leads directly to an incorrect behavior. For example, a du-anomaly does not always have direct effects; the program could still run properly. The question arises why this particular assignment is at this position in the program, just before the end of the block where the variable is valid. Usually, an exact examination of the program parts where trouble is indicated is worthwhile and further inconsistencies can be discovered.

#### 4.2.4 Execution of Control Flow Analysis

In figure 4-1, a program structure is represented as a control flow graph. In this directed graph, the statements of the program are represented with nodes. Sequences of statements are also represented with a single node because inside the sequence there can be no change in the course of program execution. If the first statement of the sequence is executed, the others are also executed.

*Control flow graph*

Changes in the course of program execution are made by decisions, such as, for example, in IF statements. If the calculated value of the condition is true, then the program continues in the part that begins with

THEN. If the condition is false, then the ELSE part is executed. Loops lead to previous statements, resulting in repeated execution of a part of the graph.

*Control flow anomalies*

Due to the clarity of the control flow graph, the sequences through the program can easily be understood and possible anomalies can be detected. These anomalies could be jumps out of a loop body or a program structure that has several exits. They may not necessarily lead to failure, but they are not in accordance with the principles of structured programming. It is assumed that the graph is not generated manually but that it is generated by a tool that guarantees an exact mapping of the program text to the graph.

If parts of the graph or the whole graph are very complex and the relations, as well as the course of events, are not understandable, then the program text should be revised, because complex sequence structures often bear a great risk of being wrong.

**Excursion:**  
**Predecessor-successor**  
**table**

In addition to graphs, a tool can generate predecessor-successor tables that show how every statement is related to the other statements. If a statement does not have a predecessor, then this statement is unreachable (so-called dead code). Thus a defect or at least an anomaly is detected. The only exceptions are the first and last statements of a program: They can legally have no predecessor or successor. For programs with several entrance and/or exit points, the same applies.

#### 4.2.5 Determining Metrics

*Measuring of quality*  
*characteristics*

In addition to the previously mentioned analyses, static analysis tools provide measurement values. Quality characteristics can be measured with measurement values, or metrics. The measured values must be checked, though, to see if they meet the specified requirements [ISO 9126]. An overview of currently used metrics can be found in [Fenton 91].

The definition of metrics for certain characteristics of software is based on the intent to gain a quantitative measure of software whose nature is abstract. Therefore, a metric can only provide statements concerning the one aspect that is examined, and the measurement values that are calculated are only interesting in comparison to numbers from other programs or program parts that are examined.

*Cyclomatic number*

In the following, we'll take a closer look at a certain metric: the →cyclomatic number (McCabe number [McCabe 76]). The cyclomatic number measures the structural complexity of program code. The basis of this calculation is the control flow graph.

For a control flow graph (G) of a program or a program part, the cyclomatic number can be computed like this:<sup>11</sup>

$$v(G) = e - n + 2$$

$v(G)$  = cyclomatic number of the graph G

$e$  = number of edges of the control flow graph

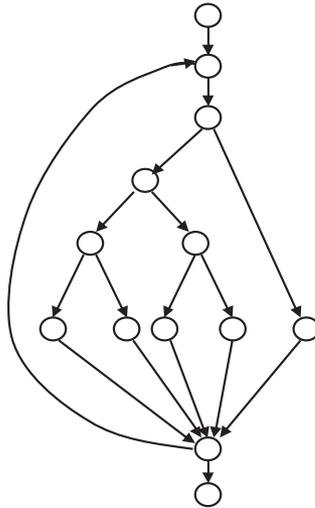
$n$  = number of nodes of the control flow graph

A program part is represented by the graph shown in figure 4-1. It is a function that can be called. Thus, the cyclomatic number can be calculated like this:

$$v(G) = e - n + 2 = 17 - 13 + 2 = 6$$

$e$  = number of edges in the graph = 17

$n$  = number of nodes in the graph = 13



**Example for computing the cyclomatic number**

**Figure 4-1**  
Control flow graph for the calculation of the cyclomatic number (identical to figure 2-2)

The value of 6 is, according to McCabe, acceptable and in the middle of the range. He assumes that a value higher than 10 cannot be tolerated and rework of the program code has to take place.

11. The original formula is  $v(G) = e - n + 2p$ , where  $p$  is the number of connected program parts. We use  $p=1$  because there is only one part that is analyzed.

*The cyclomatic number gives information about the testing effort*

The cyclomatic number can be used to estimate the testability and the maintainability of a particular program part. The cyclomatic number specifies the number of independent paths in the program part.<sup>12</sup> If 100% branch coverage (see section 5.2.2) is intended, then all these independent paths of the control flow graph have to be executed at least once. Therefore, the cyclomatic number provides important information concerning the volume of the test.

Understanding a program is essential for its maintenance.

The higher the value of the cyclomatic number, the more difficult it is to understand the flow in a certain program part.

**Excursion**

The cyclomatic number has been very much discussed since its publication. One of its drawbacks is that the complexity of the conditions, which lead to the selection of the control flow, is not taken into account. It does not matter for the calculation of the cyclomatic number whether a condition consists of several partial atomic conditions with logical operators or is a single condition. Many extensions and adaptations have been published concerning this.

### 4.3 Summary

- Several pairs of eyes see more than a single pair of eyes. This is also true in software development. This is the main principle for the reviews that are performed for checking and for improving quality. Several people inspect the documents and discuss them in a meeting and the results are recorded.
- A fundamental review process consists of the following activities: planning, kick-off, preparation, review meeting, rework, and follow-up. The roles of the participants are manager, moderator, author, reviewer, and recorder.
- There are several types of reviews. Unfortunately, the terminology is defined differently in all literature and standards.
- The walkthrough is an informal procedure where the author presents her document to the reviewers in the meeting. There is little preparation for the meeting. The walkthrough is especially suitable for small development teams, for discussing alternatives, and for educating people in the team.
- The inspection is the most formal review type. Preparation is done using checklists, there are defined entry and exit criteria, and a trained

---

12. This means all complete linearly independent paths.

---

moderator chairs the meeting. The objective of inspections is checking the quality of the document and improvement of development, the development process, and the inspection process itself.

- In the technical review, the individual reviewers' results must be given to the review leader prior to the meeting. The meeting is then prioritized by assumed importance of the individual issues. The evaluators usually have access to the specifications and other documentation only. The author can remain anonymous.
- The informal review is not based on a formal procedure. The form in which the results have to be presented is not prescribed. Because this type of review can be performed with minimal effort, its acceptance is very high, and in practice it is commonly used.
- Generally, the specific environment, i.e., the organization and project for which the review is used, determines the type of review to be used. Reviews are tailored to meet specific needs and requirements, which increases their efficiency. It is important to establish a cooperative and collaborative atmosphere among the people involved in the development of the software.
- In addition to the reviews, a lot of checks can be done for documents that have a formalized structure. These checks are called static analyses. The test object is not executed during a static analysis.
- The compiler is the most common analysis tool and reveals syntax errors in the program code. Usually, compilers provide even more checking and information.
- Analysis tools that are dependent on programming language can also show violation of standards and other conventions.
- Tools are available for detecting anomalies in the data and control flows of the program. Useful information about control and data flows is generated, which often points to parts that could contain defects.
- Metrics are used to measure quality. One such metric is the cyclomatic number, which calculates the number of independent paths in the checked program. It is possible to gain information on the structure and the testing effort.
- Generally, static analyses should be performed first, before a document is subjected to review. Static analyses provide a relatively inexpensive means to detect defects and thus make the reviews less expensive.



## 5 Dynamic Analysis – Test Design Techniques

*This chapter describes techniques for testing software by executing the test objects on a computer. It presents the different techniques, with examples, for specifying test cases and for defining test exit criteria.*

*These →test design techniques are divided into three categories: black box testing, white box testing, and experience-based testing.*

Usually, testing of software is seen as the execution of the test object on a computer. For further clarification, the phrase →dynamic analysis is used. The test object (program) is fed with input data and executed. To do this, the program must be executable. In the lower test stages (component and integration testing), the test object cannot be run alone but must be embedded into a test harness or test bed to obtain an executable program (see figure 5-1).

*Execution of the test object  
on a computer*

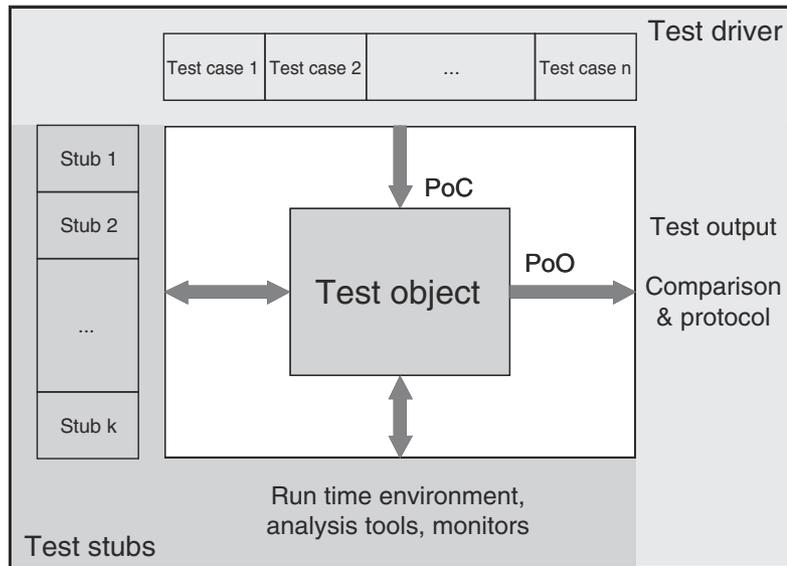
The test object will usually call different parts of the program through predefined interfaces. These parts of the program are represented by placeholders called stubs when they are not yet implemented and therefore aren't ready to be used or if they should be simulated for this particular test of the test object. Stubs simulate the input/output behavior of the part of the program that usually would be called by the test object.<sup>1</sup>

*A test bed is necessary*

---

1. In contrast to stubs, with their rudimental functionality, the →dummy or →mock-up offers additional functionality, often near the final functionality for testing purposes. A mock-up usually has more functionality than a dummy.

**Figure 5-1**  
Test bed



Furthermore, the test bed must supply the test object with input data. In most cases, it is necessary to simulate a part of the program that is supposed to call the test object. A test driver does this. Driver and stub combined establish the test bed. Together, they constitute an executable program with the test object itself.

The tester must often create the test bed, or the tester must expand or modify standard (generic) test beds, adjusting them to the interfaces of the test object. Test bed generators can be used as well (see section 7.1.4). An executable test object makes it possible to execute the dynamic test.

*Systematic approach for determining the test cases*

The objective of testing is to show that the implemented test object fulfills specified requirements as well as to find possible faults and failures. With as little cost as possible, as many requirements as possible should be checked and as many failures as possible should be found. This goal requires a systematic approach to test case design. Unstructured testing “from your gut feeling” does not guarantee that as many as possible, maybe even all, different situations supported by the test object are tested.

*Step wise approach*

The following steps are necessary to execute the tests:

- Determine conditions and preconditions for the test and the goals to be achieved.
- Specify the individual test cases.

- Determine how to execute the tests (usually chaining together several test cases).

This work can be done very informally (i.e., undocumented) or in a formal way as described in this chapter. The degree of formality depends on several factors, such as the application area of the system (for example, safety-critical software), the maturity of the development and test process, time constraints, and knowledge and skill level of the project participants, just to mention a few.

At the beginning of this activity, the test basis is analyzed to determine what must be tested (for example, that a particular transaction is correctly executed). The test objectives are identified, for example, demonstrating that requirements are met. The failure risk should especially be taken into account. The tester identifies the necessary preconditions and conditions for the test, such as what data should be in a database.

*Conditions, preconditions, and goals*

The traceability between specifications and test cases allows an analysis of the impact of the effects of changed specifications on the test, that is, the necessity for creation of new test cases and removal or change of existing ones. Traceability also allows checking a set of test cases to see if it covers the requirements. Thus, coverage can be a criterion for test exit.

*Traceability*

In practice, the number of test cases can soon reach hundreds or thousands. Only traceability makes it possible to identify the test cases that are affected by specification changes.

Part of the specification of the individual test cases is determining test input data for the test object. They are determined using the methods described in this chapter. However, the preconditions for executing the test case, as well as the expected results and expected postconditions, are necessary for determining if there is a failure (for detailed descriptions, see [IEEE 829]).

→*Test case specification*

The expected results (output, change of internal states, etc.) should be determined and documented before the test cases are executed. Otherwise, an incorrect result can easily be interpreted as correct, thus causing a failure to be overlooked.

*Determining expected result and behavior*

It does not make much sense to execute an individual test case. Test cases should be grouped in such a way that a whole sequence of test cases is executed (test sequence, test suite or test scenario). Such a test sequence is documented in the →test procedure specifications or test instructions. This document commonly groups the test cases by topic or by test objectives. Test priorities and technical and logical dependencies between the

*Test case execution*

tests and regression test cases should be visible. Finally, the test execution schedule (assigning tests to testers and determining the time for execution) is described in a  $\rightarrow$ test schedule document.

To be able to execute a test sequence, a  $\rightarrow$ test procedure or  $\rightarrow$ test script is required. A test script contains instructions for automatically executing the test sequence, usually in a programming language or a similar notation, the test script may contain the corresponding preconditions as well as instruction for comparing the actual and expected results. JUnit is an example of a framework that allows easy programming of test scripts in Java [URL: xunit].

*Black box and white box  
test design techniques*

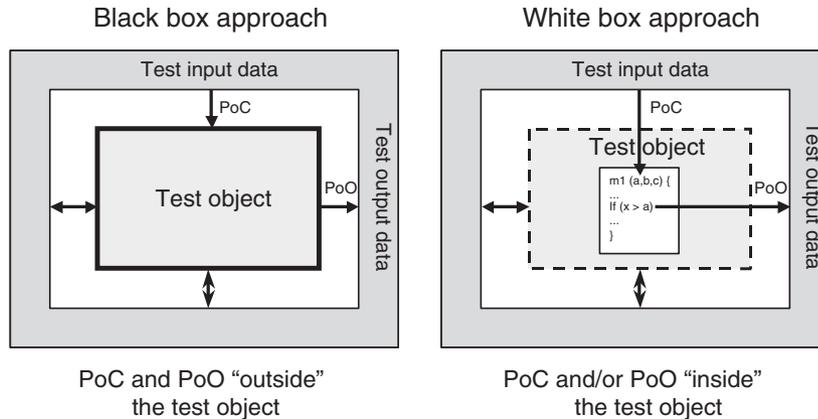
Several different approaches are available for designing tests. They can roughly be categorized into two groups: black box techniques<sup>2</sup> and white box techniques<sup>3</sup>. To be more precise, they are collectively called test case design techniques because they are used to design the respective test cases.

In black box testing, the test object is seen as a black box. Test cases are derived from the specification of the test object; information about the inner structure is not necessary or available. The behavior of the test object is watched from the outside (the  $\rightarrow$ Point of Observation, or PoO, is outside the test object). The operating sequence of the test object can only be influenced by choosing appropriate input test data or by setting appropriate preconditions. The  $\rightarrow$ Point of Control (PoC) is also located outside of the test object. Test cases are designed using the specification or the requirements of the test object. Often, formal or informal models of the software or component specification are used. Test cases can be systematically derived from such models.

In white box testing, the program text (code) is used for test design. During test execution, the internal flow in the test object is analyzed (the Point of Observation is inside the test object). Direct intervention in the execution flow of the test object is possible in special situations, such as, for example, to execute negative tests or when the component's interface is not capable of initiating the failure to be provoked (the Point of Control can be located inside the test object). Test cases are designed with the help of the program structure (program code or detailed specification) of the test object (see figure 5-2). The usual goal of white box techniques is to

- 
2. Black box techniques are also called requirements-based testing techniques
  3. White box techniques are sometimes called *glass box* or *open box* techniques because it is impossible to see through a white box. However, these terms are not used often.

achieve a specified coverage; for example, 80% of all statements of the test object shall be executed at least once. Extra test cases may be systematically derived to increase the degree of coverage.



**Figure 5-2**

*PoC and PoO at black box and white box techniques*

White box testing is also called structural testing because it considers the structure (component hierarchy, control flow, data flow) of the test object. The black box testing techniques are also called functional, specification-based, or behavioral testing techniques because the observation of the input/output behavior is the main focus [Beizer 95]. The functionality of the test object is the center of attention.

White box testing can be applied at the lower levels of the testing, i.e., component and integration test. A system test oriented on the program text is normally not very useful. Black box testing is predominantly used for higher levels of testing even though it is reasonable in component tests. Any test designed before the code is written (test-first programming, test-driven development) is essentially applying a black box technique.

Most test methods can clearly be assigned to one of the two categories. Some have elements of both and are sometimes called *gray box techniques*.

In the sections 5.1 and 5.2, black box and white box techniques are described in detail.

Intuitive and experience-based testing is usually black box testing. It is described in a special section (section 5.3) because it is not a systematic technique. This test design technique uses the knowledge and skill of people (testers, developers, users, stakeholders) to design test cases. It also uses knowledge about typical or probable faults and their distribution in the test object.

*Intuitive test case design*

## 5.1 Black Box Testing Techniques

In black box testing, the inner structure and design of the test object is unknown or not considered. The test cases are derived from the specification, or they are already available as part of the specification (“specification by example”). Black box techniques are also called specification based because they are based on specifications (of requirements). A test with all possible input data combinations would be a complete test, but this is unrealistic due to the enormous number of combinations (see section 2.1.4). During test design, a reasonable subset of all possible test cases must be selected. There are several methods to do that, and they will be shown in the following sections.

### 5.1.1 Equivalence Class Partitioning

*Input domains are divided into equivalence classes*

The domain of possible input data for each input data element is divided into  $\rightarrow$ equivalence classes (equivalence class partitioning). An equivalence class is a set of data values that the tester assumes are processed in the same way by the test object. Testing one representative of the equivalence class is considered sufficient because it is assumed that for any other input value of the same equivalence class, the test object will show the same reaction or behavior. Besides equivalence classes for correct input, those for incorrect input values must be tested as well.

**Example for equivalence class partitioning**

The example for the calculation of the dealer discount from section 2.2.2 is revisited here to clarify the facts. Remember, the program will prescribe the dealer discount. The following text is part of the description of the requirements: “For a sales price of less than \$15,000, no discount shall be given. For a price up to \$20,000, a 5% discount is given. Below \$25,000, the discount is 7%, and from \$25,000 onward, the discount is 8.5%.”

Four different equivalence classes with correct input values (called *valid* equivalence classes, or vEC, in the table) can easily be derived for calculating the discount (see table 5-1).

**Table 5-1**  
*Valid equivalence classes and representatives*

Parameter	Equivalence classes	Representative
Sales price	vEC1: $0 \leq x < 15000$	14500
	vEC2: $15000 \leq x \leq 20000$	16500
	vEC3: $20000 < x < 25000$	24750
	vEC4: $x \geq 25000$	31800

In section 2.2.2, the input values 14,500, 16,500, 24,750, 31,800 (see table 2-2) were chosen.

Every value is a representative for one of the four equivalence classes. It is assumed that test execution with input values like, for example, 13400, 17000, 22300, and 28900 does not lead to further insights and therefore does not find further failures. With this assumption, it is not necessary to execute those extra test cases. Note that tests with boundary values of the equivalence classes (for example, 15000) are discussed in section 5.1.2.

Besides the correct input values, incorrect or invalid values must be tested. Equivalence classes for incorrect input values must be derived as well, and test cases with representatives of these classes must be executed. In the example we used earlier, there are the following two invalid equivalence classes<sup>4</sup> (iEC).

Parameter	Equivalence classes	Representative
Sales price	iEC1: $x < 0$ negative, i.e., wrong sales price	-4000
	iEC2: $x > 1000000$ unrealistically high sales price <sup>a</sup>	1500800

- a. The value 1,000,000 is chosen arbitrarily. Discuss with the car manufacturer or dealer what is unrealistically high!

The following describes how to systematically derive the test cases. For every input data element that should be tested (e.g., function/method parameter at component tests or input screen field at system tests), the domain of all possible input values is determined. This domain is the equivalence class containing all valid or allowed input values. Following the specification, the program must correctly process these values. The values outside of this domain are seen as equivalence classes with invalid input values. For these values as well, it must be tested how the test object behaves.

The next step is refining the equivalence classes. If the test object's specification tells us that some elements of equivalence classes are processed differently, they should be assigned to a new (sub) equivalence class. The equivalence classes should be divided until each different requirement corresponds to an equivalence class. For every single equivalence class, a representative value should be chosen for a test case.

4. A more correct term would be *equivalence classes for invalid values* instead of *invalid equivalence class* because the equivalence class itself is not invalid, only the values of this class, referring to the specified input.

*Equivalence classes with invalid values*

**Table 5-2**  
*Invalid equivalence classes and representatives*

*Systematic development of the test cases*

*Further partitioning of the equivalence classes*

To complete the test cases, the tester must define the preconditions and the expected result for every test case.

*Equivalence classes  
for output values*

The same principle of dividing into equivalence classes can be used for the output data. However, identification of the individual test cases is more expensive because for every chosen output value, the corresponding input value combination causing this output must be determined. For the output values as well, the equivalence classes with incorrect values must not be left out.

Partitioning into equivalence classes and selecting the representatives should be done carefully. The probability of failure detection is highly dependent upon the quality of the partitioning as well as which test cases are executed. Usually, it is not trivial to produce the equivalence classes from the specification or from other documents.

*Boundaries of the  
equivalence classes*

The best test values are certainly those verifying the boundaries of the equivalence classes. There are often misunderstandings or inaccuracies in the requirements at these spots because our natural language is not precise enough to accurately define the limits of the equivalence classes. The colloquial phrase ... *less than \$15000* ... within the requirements may mean that the value 15000 is inside (EC:  $x \leq 15000$ ) or outside of the equivalence class (EC:  $x < 15000$ ). An additional test case with  $x = 15000$  may detect a misinterpretation and therefore failure. Section 5.1.2 discusses the analysis of the boundary values for equivalence classes in detail.

**Example:**  
**Equivalence class  
construction for integer  
input values**

To clarify the procedure for building equivalence classes, all possible equivalence classes for an integer input value shall be identified. The following equivalence classes result for the integer parameter `extras` of the function `calculate_price()`:

**Table 5-3**  
*Equivalence classes  
for integer input values*

Parameter	Equivalence classes
extras	$vEC_1: [MIN\_INT, \dots, MAX\_INT]^a$ $iEC_1: NaN$ (Not a Number)

- a. `MIN_INT` and `MAX_INT` each describe the minimum and maximum integer number that the computer can represent. These may vary depending on the hardware.

Notice that the domain is limited on a computer by the computer's maximum and minimum values, contrary to plain mathematics. Using values outside the computer domain often leads to failures because such exceptions are not caught correctly.

The equivalence class for incorrect values is derived from the following consideration: Incorrect values are numbers that are greater or smaller than the range of the applicable interval or every nonnumeric value.<sup>5</sup> If it is assumed that the program's reaction on an incorrect value is always the same (e.g., an exception handling that delivers the error code `NOT_VALID`), then it is sufficient to map all possible incorrect values on one common equivalence class (named NaN for *Not a Number* here). Floating-point numbers are part of this equivalence class because it is expected that the program displays an error message to inputs such as 3.5. In this case, the equivalence class partitioning method does not require any further subdivision because the same reaction is expected for every wrong input. However, an experienced tester will always include a test case with a floating-point number in order to determine if the program rounds the number and then uses the corresponding integer number for its computation. The basis for this additional test case is thus experience-based testing (see section 5.3).

Because negative and positive values are usually handled differently, it is sensible to further partition the valid equivalence class (cEV1). Zero is also an input, which often leads to failure.

Parameter	Equivalence classes	Representatives
extras	vEC <sub>1</sub> : [MIN_INT, ..., 0] <sup>a</sup> vEC <sub>2</sub> : [0, ..., MAX_INT] iEC <sub>1</sub> : NaN (Not a Number)	-123 654 "f"

- a. <sup>a</sup> Specifies an open interval until just below the given value, but not including it. The definition [MIN\_INT, ..., -1] is equivalent because we deal with integer numbers in this case.

The representative was chosen relatively arbitrarily from the three equivalence classes. Additionally, we should test the boundary values (see section 5.1.2) of the corresponding equivalence classes: MIN\_INT, -1, 0, MAX\_INT. For the equivalence classes of the invalid values, no boundary values can be given.

Thus, using equivalence partitioning and including the boundary values for the integer parameter `extras` results in the following seven values to be tested:

```
{"f", MIN_INT, -123, -1, 0, 654, MAX_INT}.
```

For each of these inputs, the predicted outputs or reactions of the test object must be defined, in order to decide after running the test if there was a failure.

**Table 5-4**

*Equivalence classes and representative values for integer inputs*

5. If, and which, incorrect values are found by the compiler depends on the chosen programming language and the compiler and runtime system chosen. This may happen when calling the test driver. In our example we assume that the compiler does not recognize incorrect parameter values. Thus, their processing must be checked during dynamic testing.

*Equivalence classes of inputs, which are no basic data types*

For the integer input data of the example, it is very easy to determine equivalence classes and the corresponding representative test values. Besides the basic data types, data structures and sets of objects can also occur. It must then be decided in each case with which representative values to execute the test case.

**Example for input values to be selected from a set**

The following example should clarify this: A potential customer can be a working person, a student, a trainee, or a retired person. If the test object needs to react differently to each kind of customer, then every possibility must be verified with an additional test case. If there is no requirement for different reactions for each person type, then one test case may be sufficient.

If the test object is the component that calculates payment options (*EasyFinance*), then four different test cases must be provided. Financing will surely be calculated differently for the different customer groups. Details must be looked up in the requirements. Each calculation must be verified by a test to check the correctness of the calculations and to find failures.

For the test of the component that handles the online configuration of the car (*VirtualShowRoom*), it may be sufficient to choose only one representative for the customer, such as, for example, a working person. It is probably not relevant if a student or a retired person configures the car. The tester should, however, be aware that if she executes the test with the input *working person* only, she would not be able to tell anything about the correctness of the car configuration for any of the other person groups.

**Hint for determining equivalence classes**

The following hints can help determine equivalence classes:

- For the inputs as well as for the outputs, identify the restrictions and conditions from the specification.
- For every restriction or condition, partition into equivalence classes:
  - If a continuous numerical domain is specified, then create one valid and two invalid equivalence classes.
  - If a number of values should be entered, then create one valid (with all possible correct values) and two invalid equivalence classes (less and more than the correct number).
  - If a set of values is specified where each value may possibly be treated differently, then create one valid equivalence class for each value of the set (containing exactly this one value) and one additional invalid equivalence class (containing all possible other values).
  - If there is a condition that must be fulfilled, then create one valid and one invalid equivalence class to test the condition fulfilled and not fulfilled.
- If there is any doubt that the values of one equivalence class are treated equally, the equivalence class should be divided further into subclasses.

## Test Cases

Usually, the test object has more than one input parameter. The equivalence class technique results in at least two equivalence classes (one valid and one invalid) for each of these parameters of the test object. Therefore, there are at least two representative values that must be used as test input for each parameter.

*Combination of the representatives*

In order to specify a test case, you must assign each parameter an input value. For this purpose, it must be decided which of the available values should be combined to form test cases. To guarantee that all test object reactions (modeled by the equivalence class division) are triggered, you must combine the input values (i.e., the representatives of the corresponding equivalence classes), using the following rules:

- The representative values of all valid equivalence classes should be combined to test cases, meaning that all possible combinations of valid equivalence classes will be covered. Any of those combinations builds a *valid test case* or a *positive test case*.
- The representative value of an invalid equivalence class shall be combined only with representatives of other *valid* equivalence classes. Thus, for every invalid equivalence class an additional *negative test case* shall be specified.

*Rules for test case design*

*Separate test of the invalid value*

The number of *valid* test cases is the product of the number of valid equivalence classes per parameter. Because of this multiplicative combination, even a few parameters can generate hundreds of *valid test cases*. Since it is seldom possible to use that many test cases, more rules are necessary to reduce the number of *valid* test cases:

*Restriction of the number of test cases*

- Combine the test cases and sort them by frequency of occurrence (typical usage profile). Prioritize the test cases in this order. That way only the *relevant* test cases (or combinations appearing often) are tested.
- Test cases including boundary values or boundary value combinations are preferred.
- Combine every representative of one equivalence class with every representative of the other equivalence classes (i.e., pairwise combinations<sup>6</sup> instead of complete combinations).

*Rules for test case restriction*

---

6. See section 4.2.5 in [Bath 08]. The pairwise combination test method is described in this book in section 5.1.4.

- Ensure that every representative of an equivalence class appears in at least one test case. This is a minimum criterion.
- Representatives of invalid equivalence classes should not be combined with representatives of other invalid equivalence classes.

*Test invalid values separately*

The representatives of invalid equivalence classes are not combined. An invalid value should only be combined with *valid* ones because an incorrect parameter value normally triggers an exception handling. This is usually independent of values of other parameters. If a test case combines more than one incorrect value, defect masking may result and only one of the possible exceptions is actually triggered and tested. When a failure appears, it is not obvious which of the incorrect values has triggered the effect. This leads to extra time and cost for failure analysis.<sup>7</sup>

**Example:**  
**Test of the DreamCar price calculation**

In the following example, the function `calculate_price()` from the VSR-Subsystem *DreamCar* serves as test object (specified in section 3.2.3). We must test if the function calculates the correct total price from its input values. We assume that the inner structure of the function is unknown. Only the functional specification of the function and the external interface are known.

```
double calculate_price (
double baseprice,    // base price of the vehicle
double specialprice, // special model addition
double extraprice,   // price of the extras
int extras,          // number of extras
double discount      // dealer's discount
)
```

*Step 1:*  
*Identifying the domain*

The equivalence class technique is used to derive the required test cases from the input parameters. First, we identify the domain for every input parameter. This results in equivalence classes for valid and invalid values for each parameter (see table 5-5).

With this technique, at least one valid and one invalid equivalence class per parameter has been derived exclusively from the interface specifications (test data generators work in a similar way; see section 7.1.2).

---

7. It is sometimes useful to combine representatives of invalid equivalence classes to produce additional test cases, thus provoking further failures.

Parameter	Equivalence classes
baseprice	vEC <sub>11</sub> : [MIN_DOUBLE, ... , MAX_DOUBLE] iEC <sub>11</sub> : NaN
specialprice	vEC <sub>21</sub> : [MIN_DOUBLE, ... , MAX_DOUBLE] iEC <sub>21</sub> : NaN
extraprice	vEC <sub>31</sub> : [MIN_DOUBLE, ... , MAX_DOUBLE] iEC <sub>31</sub> : NaN
extras	vEC <sub>41</sub> : [MIN_INT, ... , MAX_INT] iEC <sub>41</sub> : NaN
discount	vEC <sub>51</sub> : [MIN_DOUBLE, ... , MAX_DOUBLE] iEC <sub>51</sub> : NaN

**Table 5-5**

*Equivalence classes  
for integer input values*

In order to further subdivide these equivalence classes, information about the functionality of this method is needed. The functional specification delivers this information (see section 3.2.3). From this specification the following statements relevant for testing can be found:

- Parameters 1 to 3 are prices (of cars). Prices are not negative. The specification does not define any price limits.
- The value `extras` controls the discount for the supplementary equipment (10% if `extras` ≥ 3 and 15% if `extras` ≥ 5). The parameter `extras` defines the number of chosen parts of supplementary equipment and therefore it cannot be negative.<sup>8</sup> The specification does not define an upper limit for the number.
- The parameter `discount` denotes a general discount and is given as a percentage between 0 and 100. Because the specification text defines the limits for the discount for supplementary equipment as a percentage, the tester can assume that this parameter is entered as a percentage as well. Consultation with the client will otherwise clarify this matter.

These considerations are based not only on the functional specification. Rather, the analysis uncovers some “holes” in the specification. The tester fills these holes by making plausible assumptions based on application domain or general knowledge and her testing experience or by asking colleagues (testers or developers). If there is any doubt, consultation with the client is useful. The equivalence classes already defined before can be refined (partitioned into subclasses) during this analysis. The more detailed the equivalence classes are, the more precise the test. The class partition is complete when all conditions in the specification as well as conditions from the tester’s knowledge are incorporated.

*Step 2: Refine the  
equivalence classes based  
on the specification*

8. Floating-point numbers are part of the equivalence class NaN. See table 5-5 for designing equivalence classes for integer number values.

**Table 5-6**

Further partitioning of the equivalence classes of the parameter of the function *Calculate\_price()* with representatives

Parameter	Equivalence classes	Representatives
baseprice	vEC <sub>11</sub> : [0, ... , MAX_DOUBLE]	20000.00
	iEC <sub>11</sub> : [MIN_DOUBLE, ... , 0 <sup>a</sup>	-1.00
	iEC <sub>12</sub> : NaN	"abc"
specialprice	vEC <sub>21</sub> : [0, ... , MAX_DOUBLE]	3450.00
	iEC <sub>21</sub> : [MIN_DOUBLE, ... , 0[	-1.00
	iEC <sub>22</sub> : NaN	"abc"
extraprice	vEC <sub>31</sub> : [0, ... , MAX_DOUBLE]	6000.00
	iEC <sub>31</sub> : [MIN_DOUBLE, ... , 0[	-1.00
	iEC <sub>32</sub> : NaN	"abc"
extras	vEC <sub>41</sub> : [0, ... , 2]	1
	vEC <sub>42</sub> : [3, 4]	3
	vEC <sub>43</sub> : [5, ... , MAX_INT]	20
	iEC <sub>41</sub> : [MIN_INT, ... , 0[	-1.00
	iEC <sub>42</sub> : NaN	"abc"
discount	vEC <sub>51</sub> : [0, ... , 100]	10.00
	iEC <sub>51</sub> : [MIN_DOUBLE, ... , 0[	-1.00
	iEC <sub>52</sub> : ]100, ... , MAX_DOUBLE]	101.00
	iEC <sub>53</sub> : NaN	"abc"

a. 0[ means approaching, but not including zero.

The result: Altogether, 18 equivalence classes are produced, 7 for correct/valid parameter values and 11 for incorrect/invalid ones.

*Step 3:*  
*Select representatives*

To get input data, one representative value must be chosen for every equivalence class. According to equivalence class theory, any value of an equivalence class can be used. In practice, perfect decomposition is seldom done. Due to an absence of detailed information, lack of time, or just lack of motivation, the decomposition is aborted at a certain level. Several equivalence classes might even (incorrectly) overlap.<sup>9</sup> Therefore, one must remember that there could be values inside an equivalence class where the test object could react differently. Usage frequencies of different values may also be important.

Hence, in the example, the values for the valid equivalence classes are selected to represent plausible values and values that will probably often appear in practice. For invalid equivalence classes, possible values with low complexity are chosen. The selected values are shown in table 5-6.

*Step 4:*  
*Combine the test cases*

The next step is to combine the values to test cases. Using the previously given rules, we get  $1 \times 1 \times 1 \times 3 \times 1 = 3$  *valid* test cases (by combining the representatives of the valid equivalence classes) and  $2 + 2 + 2 + 2 + 3 = 11$  *negative* test cases (by separately testing representatives of every invalid class). In total, 14 test cases result from the 18 equivalence classes (table 5-7).

9. The ideal case is that the identified classes (like equivalence classes in mathematics) are not overlapping (disjoint). This should be strived for, but it's not guaranteed by the partitioning technique.

Test case	Parameter					result
	baseprice	special price	extraprice	extras	discount	
1	20000.00	3450.00	6000.00	1	10.00	27450.00
2	20000.00	3450.00	6000.00	3	10.00	26850.00
3	20000.00	3450.00	6000.00	20	10.00	26550.00
4	-1.00	3450.00	6000.00	1	10.00	NOT_VALID
5	"abc"	3450.00	6000.00	1	10.00	NOT_VALID
6	20000.00	-1.00	6000.00	1	10.00	NOT_VALID
7	20000.00	"abc"	6000.00	1	10.00	NOT_VALID
8	20000.00	3450.00	-1.00	1	10.00	NOT_VALID
9	20000.00	3450.00	"abc"	1	10.00	NOT_VALID
10	20000.00	3450.00	6000.00	-1.00	10.00	NOT_VALID
11	20000.00	3450.00	6000.00	"abc"	10.00	NOT_VALID
12	20000.00	3450.00	6000.00	1	-1.00	NOT_VALID
13	20000.00	3450.00	6000.00	1	101.00	NOT_VALID
14	20000.00	3450.00	6000.00	1	"abc"	NOT_VALID

**Table 5-7**

Further partitioning of the equivalence classes of the parameter test cases of the function *Calculate\_price()*

For the valid equivalence classes, the same representative values were used to ensure that only the variance of one parameter triggers the reaction of the test object.

Because four out of five parameters have only one valid equivalence class, only a few *valid* test cases result. There is no reason to reduce the number of test cases any further.

After the test inputs have been chosen, the expected outcome must be identified for every test case. For the negative tests this is easy: The expected result is the corresponding error code or message. For the *valid* test cases, the expected outcome must be calculated (for example, by using a spreadsheet).

### Definition of Test Exit Criteria

A test exit criterion for the test by equivalence class partitioning can be defined as the percentage of executed equivalence classes with respect to the total number of specified equivalence classes:

$$\text{EC-coverage} = (\text{number of tested EC} / \text{total number of EC}) \times 100\%$$

In the example, 18 equivalence classes have been defined, but only 15 have been executed in the chosen test cases. Then the equivalence class coverage is 83%.

$$\text{EC-coverage} = (15/18) \times 100\% = 83.33\%$$

**Example:**  
**Equivalence class**  
**coverage**

All 18 equivalence classes are contained with at least one representative each in these 14 test cases (table 5-7). Thus, executing all 14 test cases achieves 100% equivalence class coverage. If the last three test cases are left out, for example due to time limitations (i.e., only 11 instead of 14 test cases are executed), all three invalid equivalence classes for the parameter discount are not tested and the coverage will be 15/18 (for example, 83.33%).

---

*Degree of coverage defines*  
*test comprehensiveness*

The more thoroughly a test object should be tested, the higher you should plan the intended coverage. Before test execution, the predefined coverage serves as a criterion for deciding when the testing is sufficient, and after test execution, it serves as verification if the required test intensity has been achieved.

If, in the previous example, the intended coverage for equivalence classes is defined as 80%, then this can be achieved with only 14 of the 18 tests. The test using equivalence class partitioning can be finished after 14 test cases. Thus, test coverage is a measurable criterion for ending testing.

The previous example also shows how critical it is to identify the equivalence classes. If the equivalence classes have not been identified completely, then fewer representative values will be chosen for designing test cases, and fewer test cases will result. A high coverage is achieved, but it has been calculated based on an incorrect total number of equivalence classes. The supposed good result does not reflect the actual intensity of the testing. Test case identification using equivalence class partitioning is only as good as the analysis it is based on.

### **The Value of the Technique**

Equivalence class partitioning is a systematic technique. It contributes to a test where specified conditions and restrictions are not overlooked. The technique also reduces the amount of unnecessary test cases. Unnecessary test cases are the ones that have data from the same equivalence classes and therefore result in equal behavior of the test object.

Equivalence classes cannot be determined only for inputs and outputs of methods and functions. They can also be prepared for internal values and states, time-dependent values (for example, before or after an event), and interface parameters. The method can thus be used in any test level.

However, only single input or output conditions are considered. Possible dependencies or interactions between conditions are ignored. If they

are considered, this is very expensive, but it can be done through further partitioning of the equivalence classes and by specifying appropriate combinations. This kind of combination testing is also called *domain analysis*.

However, in combination with fault-oriented techniques, like boundary value analysis, equivalence class partitioning is a powerful technique.

### 5.1.2 Boundary Value Analysis

→Boundary value analysis delivers a very reasonable addition to the test cases that have been identified by equivalence class partitioning. Faults often appear at the boundaries of equivalence classes. This happens because boundaries are often not defined clearly or are misunderstood. A test with boundary values usually discovers failures. The technique can be applied only if the set of data in one equivalence class is ordered and has identifiable boundaries.

*A reasonable extension*

Boundary value analysis checks the *borders* of the equivalence classes. On every border, the exact boundary value and both nearest adjacent values (inside and outside the equivalence class) are tested. The minimal possible increment in both directions should be used. For floating-point data, this can be the defined tolerance. Therefore, three test cases result from every boundary. If the upper boundary of one equivalence class equals the lower boundary of the adjacent equivalence class, then the respective test cases coincide as well.

In many cases there does not exist a “real” boundary value because the boundary value belongs to an equivalence class. In such cases, it can be sufficient to test the boundary with two values: one value that is just inside the equivalence class and another value that is just outside the equivalence class.

---

For computing the discount on the sales price (table 5-1), four valid equivalence classes were determined and corresponding values chosen for testing the classes. Equivalence classes 3 and 4 are specified with  $vEC3: 20000 < x \leq 25000$  and  $vEC4: x \geq 25000$ . For testing the common boundary of the two equivalence classes (25000), the values 24999 and 25000 are chosen (to simplify the situation, it is assumed that only whole dollars are possible). The value 24999 lies in  $vEC3$  and is the largest possible value in that equivalence class. The value 25000 is the least possible value in  $vEC4$ . The values 24998 and 25001 do not give any more information because they are further inside their corresponding equivalence classes. Thus, when are the values 24999 and 25000 sufficient and when should we additionally use the value 25001?

**Example:**  
**Boundary values for discount**

*Two or three tests*

It can help to look at the implementation. The program will probably contain the  $\rightarrow$  instruction if  $(x < 25000)$ .<sup>10</sup> Which test cases could find a wrong implementation of this condition? The test values 24999, 25000, and 25001 generate the truth-values true, false, and false for the IF statement and the corresponding program parts are executed. Test value 25001 does not seem to add any value because test value 25000 already generates the truth-value false (and thus the change to the neighbor equivalence class). Wrong implementation of the statement if  $(x \leq 25000)$  leads to the truth-values true, true, and false. Even here, a test with the value 25001 does not lead to any new results and can thus be omitted, because the test with value 25000 will lead to a failure and thus find the fault. Only a totally wrong implementation stating, for example, if  $(x <> 25000)$  and the truth-values true, false, and true can be found with test case value 25001. The values 24999 and 25000 deliver the expected results, that is, the same ones as with the correct implementation.

**Hint**

Wrong implementation of the instruction in if  $(x > 25000)$  with false, false, and true and in if  $(x \geq 25000)$  with false, true, and true results in two or three differences between actual and expected result and can be found by test cases with the values 24999 and 25000.

To illustrate the facts, table 5-8 shows the different conditions and the truth-values of the corresponding boundary values.

**Table 5-8**

*Table with three boundary values to test the condition*

Implemented condition	24999	25000	25001	Remark
$X < 25000$ (correct)	True	False	False	Expected result
$X \leq 25000$	True	<b>True</b>	False	25000 finds the fault
$X <> 25000$	True	False	<b>True</b>	25001 find the fault
$X > 25000$	<b>False</b>	False	<b>True</b>	24999 and 25001 find the fault
$X \geq 25000$	<b>False</b>	<b>True</b>	<b>True</b>	All three values find the fault
$X == 25000$	<b>False</b>	<b>True</b>	False	24999 and 25000 find the fault

It should be decided when a test with only two values is considered enough or when it is beneficial to test the boundary with three values. The wrong query in the example program, implemented as if  $(x <> 25000)$ , can be

10. If the programmer has written if  $(x \leq 24999)$ , there will be no semantic difference from if  $(x < 25000)$ . However, the boundary values determined from analyzing the specification (24999, 25000, and 25001) do not test the implemented statement if  $(x \leq 24999)$  completely. Incorrectly implementing if  $(x == 24999)$  would give the same result (true, false, false) for the three values. A code review could in this case find the discrepancy between specification and code.

found in a code review because it does not check the boundary of a value area if ( $x < 25000$ ) but instead checks whether two values are unequal. However, this fault can easily be overlooked. Only with a boundary value test with three values can all possible wrong implementations of boundary conditions be found.

A test involving an integer input value (see section 5.1.1) produces 5 new test cases, giving us a total of 12 test cases with the following input values:

```
{ "f",
  MIN_INT-1, MIN_INT, MIN_INT+1,
  -123,
  -1, 0, 1,
  654,
  MAX_INT-1, MAX_INT, MAX_INT+1 }
```

The test case with the input value -1 tests the maximum value of the equivalence class EC1:  $[MIN\_INT, \dots 0[$ . This test case also verifies the smallest deviation from the lower boundary (0) of the equivalence class EC2:  $[0, \dots, MAX\_INT]$ . Seen from EC2, the value lies outside this equivalence class. Note that values above the uppermost boundary as well as beneath the lowermost boundary cannot always be entered due to technical reasons.

Only test values for the input variable are given in this example. To complete the test cases for each of the 12 values, the expected behavior of the test object and the expected outcome must be specified using the test oracle. Additionally, the applicable pre- and postconditions are necessary.

Here too we have to decide if the test cost is justified, and every boundary with the adjacent values must be tested with extra test cases. Test cases with values of equivalence classes that do not verify any boundary can be dropped. In the example, these are the test cases with the input values -123 and 654. It is assumed that test cases with values in the middle of an equivalence class do not deliver any new insight. This is because the maximum and the minimum values of the equivalence class are already chosen in some test cases. In the example these values are  $MIN\_INT + 1$ , 1, and  $MAX\_INT - 1$ .

**Example:**  
**Integer input**

*Is the test cost justified?*

For the example with the input data element *customer* given earlier, no boundaries for the input domain can be found. The input data type is discrete, that is, a set of the four elements (working person, student, trainee, and retired person). Boundaries cannot be identified here. A possible order by age cannot be defined clearly.

*Boundaries do not exist  
for sets*

Of course, boundary value analysis can also be applied for output equivalence classes.

**Test Cases**

Analogous to the test case determination in equivalence class partition, the valid boundaries inside an equivalence class may be combined as test cases. The invalid boundaries must be verified separately and cannot be combined with other invalid boundaries.

As described in the previous example, values from the middle of an equivalence class are, in principle, not necessary if the two boundary values in an equivalence class are used for test cases.

**Example: Boundary values for `calculate_price()`**

Table 5-9 lists the boundary values for the valid equivalence classes for verification of the function `calculate_price()`.

**Table 5-9**  
Boundaries of the parameters of the function `calculate_price()`

Parameter	Lower boundary value [Equivalence class] Upper boundary value
<b>baseprice</b>	$0-\delta^a$ , $[0, 0+\delta, \dots, \text{MAX\_DOUBLE}-\delta, \text{MAX\_DOUBLE}]$ , $\text{MAX\_DOUBLE}+\delta$
<b>specialprice</b>	Same values as baseprice
<b>extraprice</b>	Same values as baseprice
	-1, $[0, 1, 2]$ , 3 2, $[3, 4]$ , 5 4, $[5, 6, \dots, \text{MAX\_INT}-1, \text{MAX\_INT}]$ , $\text{MAX\_INT}+1$
<b>discount</b>	$0-\delta$ , $[0, 0+\delta, \dots, 100-\delta, 100]$ , $100+\delta$

- a. The accuracy considered here depends on the problem (for example, a given tolerance) and the number representation of the computer.

Considering only those boundaries that can be found inside equivalence classes, we get  $4 + 4 + 4 + 9 + 4 = 25$  boundary-based values. Of these, two (extras: 1, 3) are already tested in the original equivalence class partitioning in the example before (test cases 1 and 2 in table 5-7). Thus, the following 23 boundary values must be used for new test cases:

```
baseprice:    0.00, 0.0111, MAX_DOUBLE-0.01, MAX_DOUBLE
specialprice: 0.00, 0.01, MAX_DOUBLE-0.01, MAX_DOUBLE
extraprice:   0.00, 0.01, MAX_DOUBLE-0.01, MAX_DOUBLE
extras:       0, 2, 4, 5, 6, MAX_INT-1, MAX_INT
discount:     0.00, 0.01, 99.99, 100.00
```

As all values are valid boundaries, they can be combined into test cases (table 5-10).

---

11. For the test cases, 0.01 was assumed to be precise enough.

The expected results of a boundary value test are often not clearly visible from the specification. The experienced tester must then define reasonable expected results for her test cases:

- Test case 15 verifies all valid lower boundaries of equivalence classes of the parameters of `calculate_price()`. The test case doesn't seem to be very realistic.<sup>12</sup> This is because of the imprecise specification of the functionality, where no lower and upper boundaries are specified for the parameters (see below).<sup>13</sup>
- Test case 16 is analogous to test case 15, but here we test the precision of the calculation.<sup>14</sup>
- Test case 17 combines the next boundaries from table 5-9. The expected result is rather speculative with a discount of 99.99%. A look into the specification of the method `calculate_price()` shows that the prices are added. Thus, it makes sense to check the maximal values individually. Test cases 18 to 20 do this. For the other parameters, we use the values from test case 1 (table 5-7). Further sensible test cases result when the values of the other parameters are set to 0.00, in order to check if maximal values without further addition are handled correctly and without overflow.
- Analogous to test cases 17 to 20, test cases for `MAX_DOUBLE` should be run.
- For the still-not-tested boundary values (`extras = 5, 6, MAX_INT-1, MAX_INT` and `discount = 100.00`), more test cases are needed.

Boundary values outside the valid equivalence classes are not used here.

---

The example shows the detrimental effect of imprecise specifications on the test.<sup>15</sup> If the tester communicates with the customer before determining the test cases, and the value ranges of the parameters can be specified more precisely, then the test will be less expensive. This is shown here.

- 
12. Remark: A test with 0.00 for the base price is reasonable, but it should be done in system testing because for this input value, `calculate_price()` is not necessarily responsible for processing it.
  13. The dependence between the number of extras and extra price (if no extras are given, a price should not be displayed) cannot be checked through equivalence partitioning or boundary value analysis. This requires the use of cause-and-effect analysis (see section 5.1.4 and [Myers 79]).
  14. In order to exactly check the rounding precision, values like, for example, 0.005 are needed.
  15. And definitely for programming, too.

Testcase	Parameter					
	baseprice	specialprice	extraprice	extras	discount	result
15	0.00	0.00	0.00	0	0.00	0.00
16	0.01	0.01	0.01	2	0.01	0.03
17	MAX_DOUBLE-0.01	MAX_DOUBLE-0.01	MAX_DOUBLE-0.01	4	99.99	>MAX_DOUBLE
18	MAX_DOUBLE-0.01	3450.00	6000.00	1	10.00	>MAX_DOUBLE
19	20000.00	MAX_DOUBLE-0.01	6000.00	1	10.00	>MAX_DOUBLE
20	20000.00	3450.00	MAX_DOUBLE-0.01	1	10.00	>MAX_DOUBLE
...						

**Table 5-10**

Further test cases for the function `calculate_price()`

Early test planning—  
already during  
specification—pays off

The customer has given the following information:

- The base price is between 10000 and 150000.
- The extra price for a special model is between 800 and 3500.
- There are a maximum of 25 possible extras, whose prices are between 50 and 750.
- The dealer discount is maximum 25%.

After specifying the equivalence classes, the following valid boundary values result for the parameters:

baseprice: 10000.00, 10000.01, 149999.99, 150000.00  
specialprice: 800.00, 800.01, 3499.99, 3500.00  
extraprice: 50.00, 50.01, 18749.99, 18750.00<sup>16</sup>  
extras: 0, 1, 2, 3, 4, 5, 6, 24, 25  
discount: 0.00, 0.01, 24.99, 25.00

All these values may be freely combined to test cases. One test case is needed for each value outside the equivalence classes. The following values must be considered:

baseprice: 9999.99, 150000.01  
specialprice: 799.99, 3500.01  
extraprice: 49.99, 18750.01  
extras: -1, 26  
discount: -0.01, 25.01

16. The maximum price for extra items cannot be specified exactly because the dependence between the number of extras and the total price cannot be considered. We used the value  $25 \times 750 = 18750$ . An extra price of 0 was not included as a further boundary value because the dependency of the number of extras and the total value of the extras cannot be checked with equivalence class partitioning or boundary value analysis.

Thus, we see that a more precise specification results in fewer test cases and clear prediction of the results.

Adding the boundary values for the machine (`MAX_DOUBLE`, `MIN_DOUBLE`, etc.) is a good idea. This will detect problems with hardware restrictions.

As discussed earlier, it must be decided if it is sufficient to test a boundary with two instead of three test data values. In the following hints, we assume that two test values are sufficient because there has been a code review and possible totally wrong value area checks have been found.

- 
- For an input domain, the boundaries and the adjacent values outside the domain must be considered. Domain:  $[-1.0; +1.0]$ , test data:  $-1.0$ ,  $+1.0$  and  $-1.001$ ,  $+1.001$ .<sup>17</sup>
  - If an input file has a restricted number of data records (for example, between 1 and 100), the test values should be 1, 100 and 0, 101.
  - If the *output* domains serve as the basis, then this is the way to proceed: The output of the test object is an integer value between 500 and 1000. Test outputs that should be achieved: 500, 1000, 499, 1001. Indeed, it can be difficult to identify the respective input test data to achieve exactly the required outputs. Generating the invalid outputs may even be impossible, but you may find defects by thinking about it.
  - If the permitted number of output values is to be tested, proceed just as with the number of input values: If outputs of 1 to 4 data values are allowed, the test outputs to produce are 1, 4 as well as 0 and 5 data values.
  - For ordered sets, the first element and the last element are of special interest for the test.
  - If complex data structures are given as input or output (for instance, an empty list or zero), tables can be considered as boundary values.
  - For numeric calculations, values that are close together, as well as values that are far apart, should be taken into consideration as boundary values.
  - For invalid equivalence classes, boundary value analysis is only useful when different exception handling for the test object is expected, depending on an equivalence class boundary.
  - Additionally, extremely large data structures, lists, tables, etc. should be chosen. For example, you should exceed buffer, file, or data storage boundaries, in order to check the behavior of the test object in extreme cases.
  - For lists and tables, empty and full lists and the first and last elements are of interest because they often show failures due to incorrect programming (*Off-by-one problem*).
- 

**Hint for test case design  
by boundary analysis**

---

17. The precision to be chosen depends on the specified problem.

### Definition of the Test Exit Criteria

Analogous to the test completion criterion for equivalence class partition, an intended coverage of the boundary values (BVs) can also be predefined and calculated after execution of the tests.

$$\text{BV-Coverage} = (\text{number of tested BV} / \text{total number of BV}) \times 100\%$$

Notice that the boundary values, as well as the corresponding adjacent values above and below the boundary, must be counted. However, only differing values are used for the calculation. Overlapping values of adjacent equivalence classes are counted as only one boundary value because only one test case with the respective input value is used.

### The Value of the Technique

*In combination with  
equivalence class  
partitioning*

Boundary value analysis should be used together with equivalence class partitioning because faults can be found more often at the boundaries of the equivalence classes than far inside the classes. It makes sense to combine both techniques, but the technique still allows enough freedom in selecting the concrete test data.

The technique requires a lot of creativity to define appropriate test data at the boundaries. This aspect is often ignored because the technique appears to be very easy, even though determining the relevant boundaries is not at all trivial.

### 5.1.3 State Transition Testing

*Consider history*

In many systems, not only the current input but also the history of execution or events or inputs influences computation of the outputs and how the system will behave. History of system execution needs to be taken into account. To illustrate the dependence on history,  $\rightarrow$ state diagrams are used. They are the basis for designing the test ( $\rightarrow$ state transition testing).

The system or test object starts from an initial state and can then come into different states. Events trigger state changes or transitions. An event may be a function invocation. State transitions can involve actions. Besides the initial state, the other special state is the end state.  $\rightarrow$ Finite state machines, state diagrams, and state transition tables model this behavior.

*Definition of a finite state  
machine*

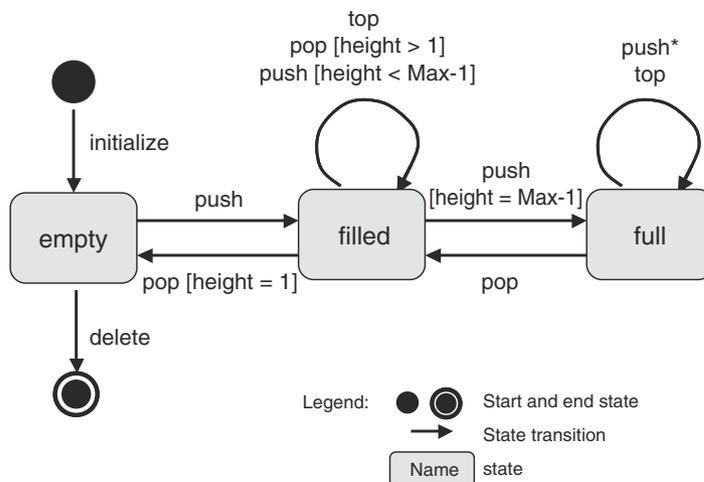
A finite state machine is formally defined as follows: An abstract machine for which the number of states and input symbols are both finite

and fixed. A finite state machine consists of states (nodes), transitions (links), inputs (link weights), and outputs (link weights). There are a finite number of internal configurations, called states. The state of a system implicitly contains the information that has resulted from the earlier inputs and that is necessary to find the reaction of the system to new inputs.

Figure 5-3 shows the popular example of a stack. The stack—for example, a dish stack in a heating device—can be in three different states: empty, filled, and full.

The stack is “empty” after initializing where the maximum height (Max) is defined (current height = 0). By adding an element to the stack (calling the function push), the state changes to “filled” and the current height is incremented. In this state further elements can be added (push, increment height) as well as withdrawn (call of the function pop, decrement height). The uppermost element can also be displayed (call of the function top, height unchanged). Displaying does not alter the stack itself and therefore does not remove any element. If the current height is one less than the maximum (height = Max - 1) and one element is added to the stack (push), then the state of the stack changes from “filled” to “full.” No further element can be added. The condition (Max - 1) is described as the *guard* for the transition between the initial and the resulting state. Appropriate guards are illustrated in figure 5-3. If one element is removed (pop) from a stack in the “full” state, the state is changed back from “full” to “filled.” A transition from “filled” to “empty” happens only if the stack consists of just one element, which is removed (pop). The stack can only be deleted in the “empty” state.

**Example:**  
**Stack**



**Figure 5-3**  
State diagram of a stack

Depending upon the specification, you can define which functions (push, pop, top, etc.) can be called for which state of the stack. You must still clarify what happens when an element is added to a “full” stack (push\*). The function must work differently from the case of a just-“filled” stack. Thus, the functions must behave differently depending on the state of the stack. The state of the test object is a decisive element and must be taken into account when testing.<sup>18</sup>

*A possible concrete test case*

Here is a possible test case with pre- and postconditions for a stack that may store text strings:

Precondition: Stack is initialized, state is “empty”

Input: Push (“hello”)

Expected reaction: The stack contains “hello”

Postcondition: State of the stack is “filled”

Further functions of the stack (showing the current level, showing the maximum level, enquiry if the stack is empty, etc.) are not considered in this example because they do not change the state of the stack.

*The test object in state  
transition testing*

In state transition testing, the test object can be a complete system with different system states as well as a class in an object-oriented system with different states. Whenever the input history leads to differing behavior, a state transition test must be applied.

***Further test cases for the  
stack example***

Different levels of test intensity can be defined for a state transition test. A minimum requirement is to get to all possible states. In the stack example, these states are empty, filled, and full.<sup>19</sup> With an assumed maximum height of 4, all three states are reached after calling the following functions:

Test case 1:<sup>20</sup> initialize [empty], push [filled], push, push, push [full].

Yet, even not all the functions of the stack have been called in this test.

Another requirement for the test is to invoke all functions. With the same stack as before, the following sequence of function calls is sufficient for compliance with this requirement:

Test case 2: initialize [empty], push [filled], top, pop [empty], delete.

However, in this sequence, not all the states have been reached.

18. Calling top and pop in the state “empty” have not been specified in the diagram (fig 5-3). This was done on purpose. They will first be taken into account in the extended state transition tree (see figure 5-5).

19. To keep the test effort small, the maximum height of the stack should be not too high because the push function must be called a corresponding number of times to get to the “full” state.

20. The following test cases are simplified to make them easy to understand.

A state transition test should execute all specified functions of a state at least once. Compliance between the specified and the actual behavior of the test object can thus be checked.

*Test criteria*

To identify the necessary test cases, the finite state machine is transformed into a so-called transition tree, which includes certain sequences of transitions ([Chow 78]). The cyclic state transition diagram with potentially infinite sequences of states changes to a transition tree, which corresponds to a representative number of states without cycles. With this translation, all states must be reached and all transitions of the transition diagram must appear.

*Design a transition tree*

The transition tree is built from a transition diagram this way:

1. The initial or start state is the root of the tree.
2. For every possible transition from the initial state to a following state in the state transition diagram, the transition tree receives a branch from its root to a node, representing this next state.
3. The process for step 2 is repeated for every leaf in the tree (every newly added node) until one of the following two end conditions is fulfilled:
  - The corresponding state is already included in the tree on the way from the root to the node. This end condition corresponds to one execution of a cycle in the transition diagram.
  - The corresponding state is a final state and therefore has no further transitions to be considered.

For the stack, the resulting transition tree is shown in figure 5-4.

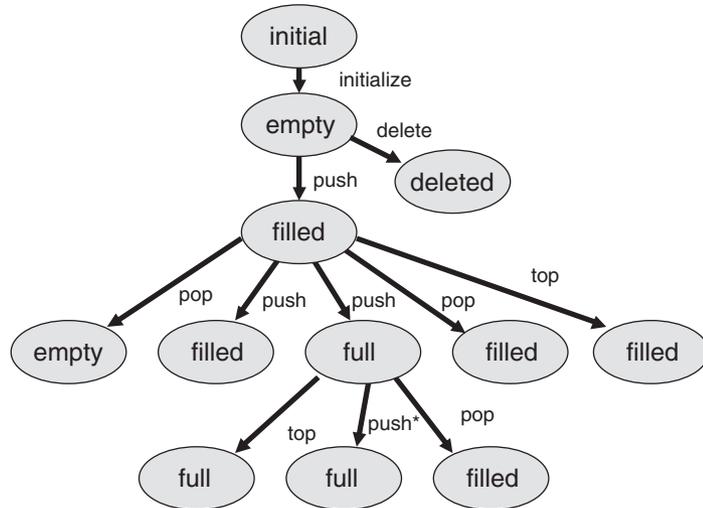
Eight different paths can be recognized from the root to each of the end nodes (leaves). Each path represents a test case, that is, a sequence of function calls. Thereby, every state is reached at least once, and every possible function is called in each state according to the specification of the state transition diagram.

However, the transition tree doesn't show the appropriate guards, but they need to be taken care of when test cases are designed.

In test case 1 (shown previously), the maximum assumed stack height is four and the guard condition for the transition from the filled to the full state when push is called is  $\text{max. height} (4) - 1 = 3$ . Three push calls are therefore necessary to pass from filled to full in the transition tree. In addition, another first push call serves to change the state from empty to filled. If no guard conditions are set in a transition tree (as in figures 5-4

and 5-5), it looks like a single push call is sufficient to move from the filled to the full state.

**Figure 5-4**  
Transition tree  
for the stack example



The transition tree shown in figure 5-4 includes all possible call *sequences* resulting from the state model shown in figure 5-3. In addition, the reaction of the state machine for wrong usage must be checked, which means that functions are called in states in which they are not supposed to be called. Here again the remark that push needs to work differently depends on the state. If push is called in the “full” state, it cannot add an element to the stack but must leave it unchanged. A message may result, but this is not the same as a fault.

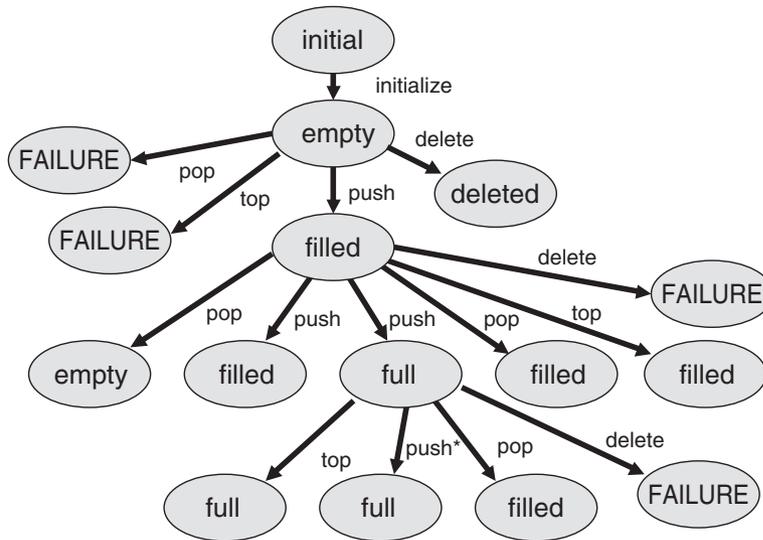
#### *Incorrect use of functions*

It is a violation of the specification if functions are called in states where they should not be used (e.g., to delete the stack while in the “full” state). A robustness test must be executed to check how the test object works when used incorrectly. It should be tested to see whether unexpected transitions appear. The test can be seen as an analogy to the test of unexpected input values.

The transition tree should be extended by adding a branch for every function from every node. This means that from every state, all the functions should be executed or at least an attempt should be made to execute them (see figure 5-5).

Producing an extended transition tree can help to find gaps in the specifications.

Here, for example, the pop and top calls that weren't present in the state diagram in figure 5-3 (i.e., that weren't specified) have been added to the state "empty." It definitely makes sense to define which reactions to expect when trying pop and top calls for an empty stack. A reasonable reaction would be, for example, an error message.



**Figure 5-5**

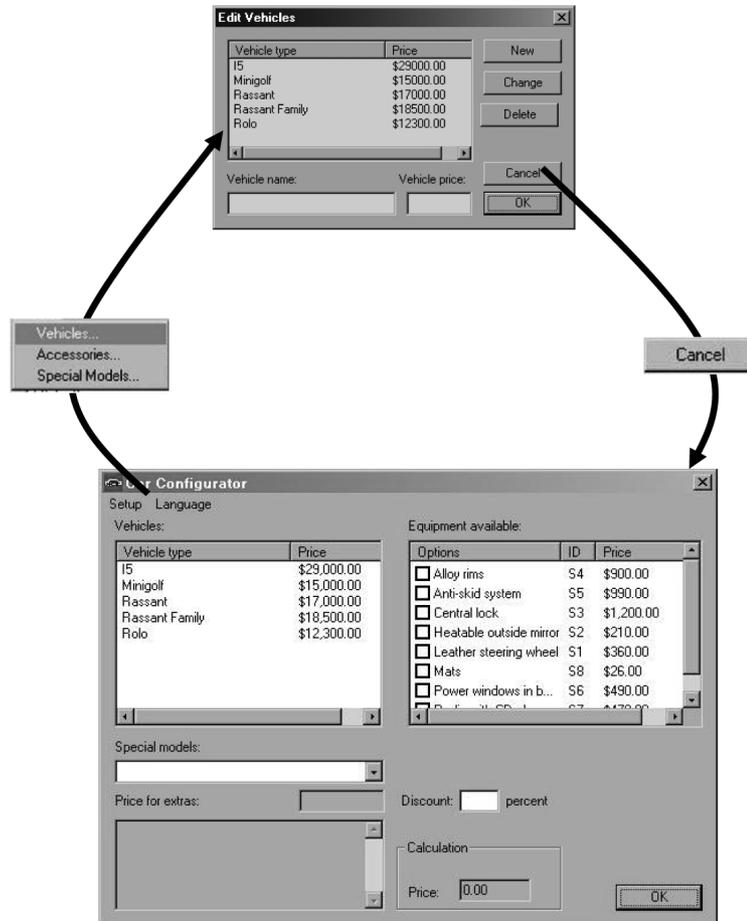
*Transition tree for the test for robustness*

State transition testing is also a good technique for system testing when testing the graphical user interface (GUI) of the test object: The GUI usually consists of a set of screens and dialog boxes; between those, the user can switch back and forth (via menu choices, an OK button, etc.). If screens and user controls are seen as states and input reactions as state transitions, then the GUI can be modeled with a state diagram. Appropriate test cases and test coverage can be identified by the state transition testing technique described earlier.

**Example:**  
**Test of DreamCar-GUI**

**Figure 5–6**  
GUI navigation as state  
graph

When testing the *DreamCar* GUI, it may look like this:



The test starts at the *DreamCar* main screen (state 1). The action<sup>21</sup> “Setup vehicles” triggers the transition into the dialog “Edit vehicle” (state 2). The action “Cancel” ends this dialog and the application returns to state 1. Inside a state we can then use “local” tests, which do not change the state. These tests then verify the built-in functionality of the accessed screen. Navigation through arbitrarily complex chains of dialogs can then be modeled after this action. The state diagram of the GUI ensures that all dialogs are included and verified in the test.

21. The two-staged menu choice is seen here as one action.

## Test Cases

To completely define a state-based test case, the following information is necessary:

- The initial state of the test object (component or system)
- The inputs to the test object
- The expected outcome or expected behavior
- The expected final state

Further, for each expected state transition of the test case, the following aspects must be defined:

- The state before the transition
- The initiating event that triggers the transition
- The expected reaction triggered by the transition
- The next expected state

It is not always easy to identify the states of a test object. Often, the state is not defined by a single variable but is rather the result from a constellation of values of several variables. These variables may be deeply hidden in the test object. Thus, the verification and evaluation of each test case can be very expensive.

- 
- Evaluate the state transition diagram from a testing point of view when writing the specification. If there are a high number of states and transitions, indicate the higher test effort and push for simplification if possible.
  - Check the specification, as well, to make sure the different states are easy to identify and that they are not the result of a multiple combination of values of different variables.
  - Check that the state variables are easy to display from the outside. It is a good idea to include functions that set, reset, and read the state for use during testing.
- 

**Hint**

## Definition of the Test Exit Criteria

Criteria for test intensity and for exiting can also be defined for state transition testing:

- Every state has been reached at least once.
- Every transition has been executed at least once.
- Every transition violating the specification has been checked.

Percentages can be defined using the proportion of test requirements that were actually executed to possible ones, similar to the earlier described coverage measures.

*Higher-level criteria*

For highly critical applications, more stringent state transition test completion criteria can be defined:

- All combination of transitions
- All transitions in any order with all possible states, including multiple executions in a row

But, achieving sufficient coverage is often not possible due to the large number of necessary test cases. Therefore, it is reasonable to set a limit to the number of combinations or sequences that must be verified.

### **The Value of the Technique**

State transition testing should be applied where states are important and where the functionality is influenced by the current state of the test object. The other testing techniques that have been introduced do not support these aspects because they do not account for the different behavior of the functions in different states.

*Especially useful for test of object-oriented systems*

In object-oriented systems, objects can have different states. The corresponding methods to manipulate the objects must then react according to what state they are in. State transition testing is therefore more important for object-oriented testing because it takes into account this special aspect of object orientation.

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

The previously introduced techniques look at the different input data independently. The input values are each considered separately for generating test cases. Dependencies among the different inputs and their effects on the outputs are not explicitly considered for test case design.

*Cause-effect graphing*

[Myers 79] describes a technique that uses the dependencies for identification of the test cases. It is known as  $\rightarrow$ cause-effect graphing. The logical relationships between the causes and their effects in a component or a system are displayed in a so-called cause-effect graph. The precondition is that it is possible to find the causes and effects from the specification. Every cause is described as a condition that consists of input values (or combinations thereof). The conditions are connected with logical operators (e.g., AND, OR, NOT). The condition, and thus its cause, can be

true or false. Effects are treated similarly and described in the graph (see figure 5-7).

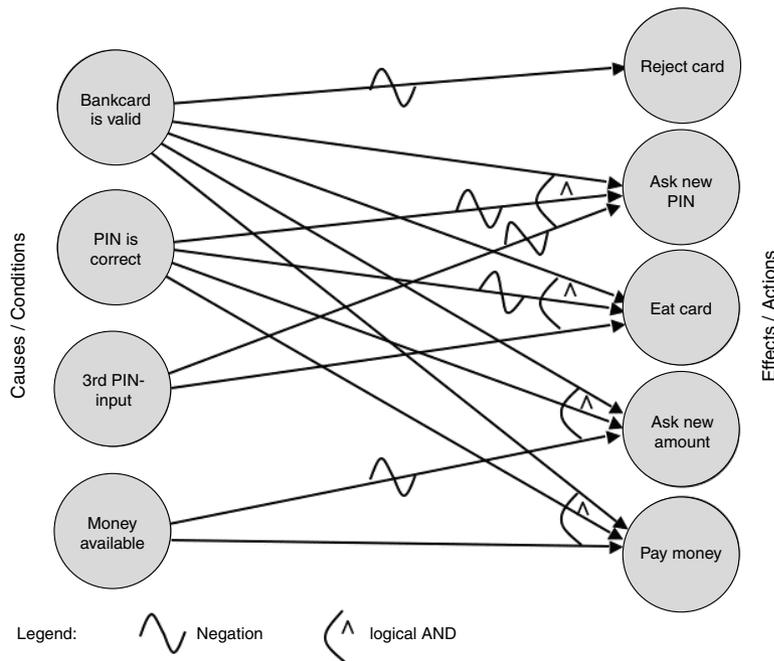
In the following example, we'll use the act of withdrawing money at an automated teller machine (ATM) to illustrate how to prepare a cause-effect graph. In order to get money from the machine, the following conditions must be fulfilled:<sup>22</sup>

- The bank card is valid.
- The PIN is entered correctly.
- The maximum number of PIN inputs is three.
- There is money in the machine and in the account.

The following actions are possible at the machine:

- Reject card.
- Ask for another PIN input.
- "Eat" the card.
- Ask for an other amount.
- Pay the requested amount of money.

Figure 5-7 shows the cause-effect graph of the example.



**Example:**  
**Cause-effect graph**  
**for an ATM**

**Figure 5-7**  
*Cause-effect graph*  
*of the ATM*

22. Note: This is not a complete description of a real automated teller machine but just an example to illustrate the technique.

The graph makes clear which conditions must be combined in order to achieve the corresponding effects.

---

The graph must be transformed into a  $\rightarrow$ decision table from which the test cases can be derived. The steps to transform a graph into a table are as follows:

1. Choose an effect.
2. Looking in the graph, find combinations of causes that have this effect and combinations that do not have this effect.
3. Add one column into the table for every one of these cause-effect combinations. Include the caused states of the remaining effects.
4. Check to see if decision table entries occur several times, and if they do, delete them.

*Test with decision tables*

The objective for a test based on decision tables is that it executes “interesting” combinations of inputs—interesting in the sense that potential failures can be detected. Besides the causes and effects, intermediate results with their truth-values may be included in the decision table.

A decision table has two parts. In the upper half, the inputs (causes) are listed; the lower half contains the effects. Every column defines the test situations, i.e., the combination of conditions and the expected effects or outputs.

In the easiest case, every combination of causes leads to one test case. However, conditions may influence or exclude each other in such a way that not all combinations make sense. The fulfillment of every cause and effect is noted in the table with a “yes” or “no.” Each cause and effect should at least once have the values “yes” and “no” in the table.

**Example:**  
**Decision table for an ATM**

Because there are four conditions (from “bank card is valid” to “money available”), there are, theoretically, 16 ( $2^4$ ) possible combinations. However, not all dependencies are taken into account here. For example, if the bank card is invalid, the other conditions are not interesting because the machine should reject the card.

An optimized decision table does not contain all possible combinations, but the impossible or unnecessary combinations are not entered. The dependencies between the inputs and the results (actions, outputs) lead to the following optimized decision table, showing the result (table 5-11).

Decision table		TC1	TC2	TC3	TC4	TC5
Conditions	Bank card valid?	N	Y	Y	Y	Y
	PIN correct?	-	N	N	Y	Y
	Third PIN attempt?	-	N	Y	-	-
	Money available?	-	-	-	N	Y
Actions	Reject card	Y	N	N	N	N
	Ask for new PIN	N	Y	N	N	N
	"Eat" card	N	N	Y	N	N
	Ask for new amount	N	N	N	Y	N
	Pay cash	N	N	N	N	Y

**Table 5-11**  
Optimized decision table  
for the ATM

Every column of this table is to be interpreted as a test case. From the table, the necessary input conditions and expected actions can be found directly. Test case 5 shows the following condition: The money is delivered only if the card is valid, the PIN is correct after a maximum of three tries, and there is money available both in the machine and in the account.

This relatively small example shows how more conditions or dependencies can soon result in large and unwieldy graphs or tables.

From a decision table, a decision tree may be derived. The decision tree is analogous to the transition tree in state transition testing in how it's used.

Every path from the root of the tree to a leaf corresponds to a test case. Every node on the way to a leaf contains a condition that determines the further path, depending on its truth-value.

### Test Cases

In a decision table, the conditions and dependencies for the inputs, the corresponding predicted outputs, and the results for this combination of inputs can be read directly from every column to form a test case. The table defines logical test cases. They must be fed with concrete data values in order to be executed, and necessary preconditions and postconditions must be defined.

*Every column is a test case*

### Definition of the Test Exit Criteria

As with the previous methods, criteria for test completion can be defined relatively easily. A minimum requirement is to execute every column in the

*Simple criteria for test exit*

decision table by at least one test case. This verifies all sensible combinations of conditions and their corresponding effects.

### **The Value of the Technique**

The systematic and very formal approach in defining a decision table with all possible combinations may show combinations that are not included when other test case design techniques are used. However, errors can result from optimization of the decision table, such as, for example, when the input and condition combinations to be considered are (erroneously) left out.

As mentioned, the graph and the table may grow quickly and lose readability when the number of conditions and dependent actions increases. Without adequate support by tools, the technique is then very difficult.

### **Pairwise Combination Testing**

This test design technique can be used when interactions between different parameters are unknown. This is the opposite of cause-effect graphing, which is designed to cover explicitly known dependencies. Pairwise combination testing has the objective of finding destructive interaction between presumably independent parameters (or parameters for which the specification does not include dependencies).

The technique starts from the equivalence class table. For every equivalence class,<sup>23</sup> a representative value is chosen. Then, every representative for one class is combined with every representative for every other class (taking into account only pairs of combinations, not higher-level combinations).

---

After installation of the *DreamCar* subsystem, three parameters must be set: the operating system (Mac, Linux, or Windows), the language (German, Norwegian, English), and the screen size (small, large). If all combinations were chosen to test this,  $3 \times 3 \times 2 = 18$  test cases would result. However, choosing pair wise combinations, we need only 9 test cases. Table 5-12 shows a possible solution.

---

23. Or even only for every valid equivalence class.

Test case #	OS	Language	Screen
1	Mac	German	small
2	Linux	German	large
3	Windows	German	large
4	Mac	Norwegian	large
5	Linux	Norwegian	small
6	Windows	Norwegian	small
7	Mac	English	large
8	Linux	English	small
9	Windows	English	Choose freely

**Table 5-12**  
*Pairwise combinations*

The solution shows that each operating system occurs with every possible language and every possible screen size. Every language also occurs with every possible screen size and with every possible operating system. Finally, every possible screen size occurs with every language and with every operating system. But not every possible triple combination (such as Mac, English, small) occurs in the test. Test case 9 is special: The combination of Windows and English is necessary, but any combinations with the screen size have already been covered in other test cases. Thus, the screen size can be freely chosen; for example, the most often occurring one can be used.

Pairwise combination tests will find any destructive interaction between supposedly independent parameters (provided the representative values chosen do this). Higher-level interactions will not necessarily be discovered.

The technique is not easy to apply manually, but tools are available [URL: pairwise].

The technique can be extended to cover higher levels of interaction.

### 5.1.5 Use-Case-Based Testing

With the increasing use of object-oriented methods for software development, the Unified Modeling Language (UML) ([URL: UML]) is used ever more frequently in practice. UML defines more than 10 graphical notations that can be used in all kinds of software development, not only object-oriented.

*UML is widely used*

→Use case testing

There are many research projects and approaches to directly derive test cases from UML diagrams and to generate these tests more or less automatically. One current issue is model-based testing.<sup>24</sup>

Requirements identification

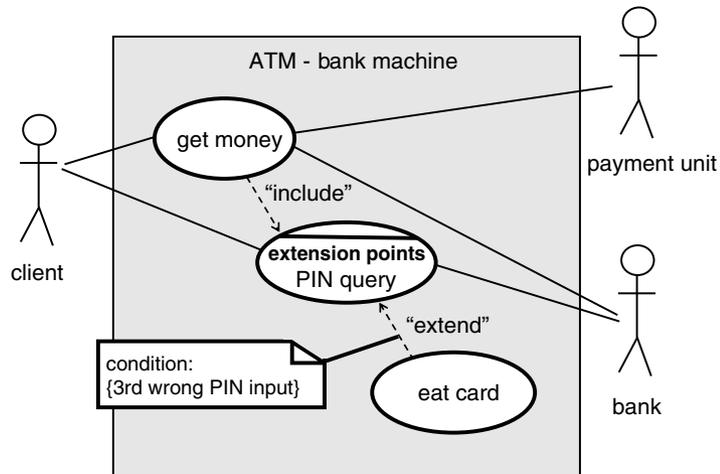
Requirements may be described as →use cases or business cases. They may be given as diagrams. The diagrams help define requirements on a relatively abstract level by describing typical user-system interactions. Testers may utilize use cases to derive test cases.

Figure 5-8 shows a use case diagram for part of the dialog with an ATM for withdrawing money.

The individual use cases in this example are “Get money,” “PIN query,” and “Eat card.” Relationships between use cases may be “include” and “extend.” “Include” conditions are always used, and “extend” connections can lead to extensions of a use case under certain conditions at a certain point (*extension point*). Thus, the “extend” conditions are not always executed; there are alternatives.

**Figure 5-8**

Use case diagram for ATM



Showing an external view

Use case diagrams mainly serve to show the external view of a system from the viewpoint of the user or to show the relation to neighboring systems. Such external connections are shown as lines to *actors* (for example, the man symbol in the figure). There are further elements in a use case diagram that are included in this discussion.

24. ISTQB [URL: ISTQB] is defining a model-based testing add-on to the Foundation Level syllabus.

For every use case, certain preconditions must be fulfilled to enable its execution. A precondition for getting money at the ATM is, for example, that the bank card is valid. After a use case is executed, there are postconditions. For example, after successfully entering the correct PIN, it is possible to get money. However, first the amount must be entered, and it must be confirmed that the money is available. Pre- and postconditions are also applicable for the flow of use cases in a diagram, that is, the path through the diagram.

*Pre- and postconditions*

Use cases and use case diagrams serve as the basis for determining test cases in use-case-based testing. As the external view is modeled, the technique is very useful for both system testing and acceptance testing. If the diagrams are used to model the interactions between different subsystems, test cases can also be derived for integration testing.

*Useful for system and acceptance testing*

The diagrams show the “normal,” “typical,” and “probable” flows and often their alternatives. Thus, the use-case-based test checks typical use of a system. It is especially important for acceptance of a system that it runs as stable as possible in “normal” use. Thus, use-case-based testing is highly relevant for the customer and user and therefore for the developer and tester as well.

*Typical system use is tested*

### **Test Cases**

Every use case has a purpose and shall achieve a certain result. Events may occur that lead to further alternatives or activities. After the execution, there are postconditions. All of the following information is necessary for designing test cases and is thus available:

- Start situation and preconditions
- Possibly other conditions
- Expected results
- Postconditions

However, the concrete input data and results for the individual test cases cannot be derived directly from the use cases. The individual input and output data must be chosen. Additionally, each alternative contained in the diagram (“extend” relation) must be covered by a test case. The techniques for designing test cases on the basis of use cases may be combined with other specification-based test design techniques.

### Definition of the Test Exit Criteria

A possible criterion is that every use case or every possible sequence of use cases in the diagram is tested at least once by a test case. Since alternatives and extensions are use cases too, this criterion also requires their execution.

### The Value of the Technique

Use-case-based testing is very useful for testing typical user-system interactions. Thus, it is best to apply it in acceptance testing and in system testing. Additionally, test specification tools are available to support this approach (section 7.1.4). “Expected” exceptions and special treatment of cases can be shown in the diagram and included in the test cases, such as, for example, entering a wrong PIN three times (see figure 5-8). However, no systematic method exists to determine further test cases for testing facts that are not shown in the use case diagram. The other test techniques, such as boundary value analysis, are helpful for this.

- Excursion** This section definitely did not describe all black box test design techniques. We'll briefly describe a few more techniques here to offer some tips about their selection. Further techniques can be found in [Myers 79], [Beizer 90], [Beizer 95], and [Pol 98].
- Syntax test* →Syntax testing describes a technique for identifying test cases that can be applied if a formal specification of the syntax of the inputs is available. Syntax testing would be used for testing interpreters of command languages, compilers, and protocol analyzers, for example. The syntax definition is used to specify test cases that cover both the compliance to and violation of the syntax rules for the inputs [Beizer 90].
- Random test* →Random testing generates values for the test cases by random selection. If a statistical distribution of the input values is given (e.g., normal distribution), then it should be used for the selection of test values. This ensures that the test cases are preferably close to reality, making it possible to use statistical models for predicting or certifying system reliability [IEEE 982], [Musa 87].
- Smoke test* The term *smoke test* is often used in software testing. A smoke test is commonly understood as a “quick and dirty” test that is primarily aimed at verifying the minimum reliability of the test object. The test is concentrated on the main functions of the test object. The output of the test is not evaluated in detail. It is checked only if the test object crashes or seriously misbehaves. A test oracle is not used, which contributes to making this test inexpensive and easy. The term *smoke test* is derived from testing old-fashioned electrical circuits because short circuits lead to smoke rising. A smoke test is often used to decide if the test object is mature enough to proceed with further testing designed with the more comprehensive test techniques. Smoke tests can also be used for first and fast tests of software updates.

### 5.1.6 General Discussion of the Black Box Technique

The basis of all black box techniques is the requirements or specifications of the system or its components and how they collaborate. Black box testing will not be able to find problems where the implementation is based on incorrect requirements or a faulty design specification because there will be no deviation between the faulty specification or design and the observed results. The test object will execute as the requirements or specifications require, even when they are wrong. If the tester is critical toward the requirements or specifications and uses “common sense”, she may find wrong requirements during test design.

*Wrong specification  
is not detected*

Otherwise, to find inconsistencies and problems in the specifications, reviews must be used (section 4.1.2).

In addition, black box testing cannot reveal extra functionality that exceeds the specifications. (Such extra functionality is often the cause of security problems.) Sometimes additional functions are neither specified nor required by the customer. Test cases that execute those additional functions are performed by pure chance if at all. The coverage criteria, which serve as conditions for test exit, are exclusively identified on the basis of the specifications or requirements. They are not based on unmentioned or assumed functions.

*Functionality that's not  
required is not detected*

The center of attention for all black box techniques is the verification of the functionality of the test object. It is indisputable that the highest priority is that the software work correctly. Thus, black box techniques should always be applied.

*Verification of the  
functionality*

## 5.2 White Box Testing Techniques

The basis for white box techniques is the source code of the test object.

*Code-based testing  
techniques*

Therefore, these techniques are often called structure-based testing techniques because they are based on the structure (of the program). They are also called →code-based testing techniques. The source code must be available, and in certain cases, it must be possible to manipulate it, that is, to add code.

The foundation of white box techniques is to execute every part of the code of the test object at least once. Flow-oriented test cases are identified, analyzing the program logic, and then they are executed. However, the expected results should be determined using the requirements or specifi-

*All code should be executed*

cations, not the code. This is done in order to decide if execution resulted in a failure.

A white box technique can focus on, for example, the statements of the test object. The primary goal of the technique is then to achieve a previously defined coverage of the statements during testing, such as, for example, to execute as many statements of the program as possible.

These are the white box test case design techniques:

- →Statement testing
- →Decision testing or →branch testing
- Testing of conditions
  - →Condition testing<sup>25</sup>
  - →Multiple condition testing
  - →Condition determination testing<sup>26</sup>
- →Path testing

The following sections describe these techniques in more detail. The ISTQB Foundation Level syllabus describes only statement and branch or decision testing.

### 5.2.1 Statement Testing and Coverage

*Control flow graph is necessary*

This analysis focuses on each *statement* of the test object. The test cases shall execute a predefined minimum quota or even all statements of the test object. The first step is to translate the source code into a control flow graph. The graph makes it easier to specify in detail the control elements that must be covered. In the graph, the statements are represented as nodes (boxes) and the control flow between the statements is represented as edges (connections). If sequences of unconditional statements appear in the program fragment, they are illustrated as one single node because execution of the first statement of the sequence guarantees that all following statements will be executed. Conditional statements (IF, CASE) and loops (WHILE, FOR), represented as control flow graphs, have more than one edge going to the exit node.

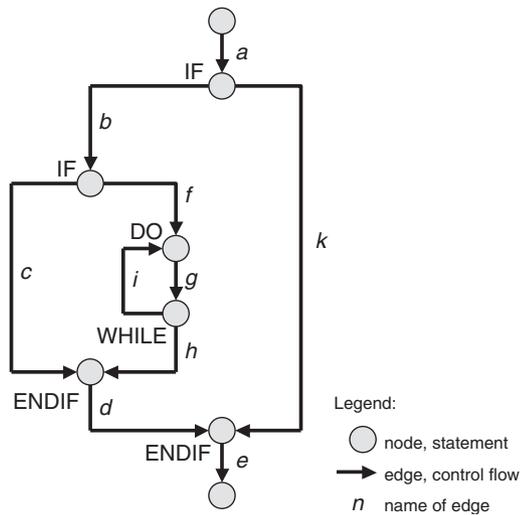
After execution of the test cases, it must be determined which statements have been executed (section 5.2.6). When the previously defined

25. Also called simple condition test or coverage.

26. Also called minimal multicondition test/coverage, modified multiple condition test/coverage, or MC/DC.

coverage level has been achieved, the test is considered to be sufficient and will therefore be terminated. Normally, all instructions should be executed because it is impossible to verify the correctness of instructions that have not been executed.

The following example will clarify how to do this. We chose a very simple program fragment for this example. It consists of only two decisions and one loop (figure 5-9).



### Example

**Figure 5-9**  
Control flow of a program fragment

### Test cases

In this example, all statements (all nodes) can be reached by a single test case. In this test case, the edges of the graph must be traversed in this order:

a, b, f, g, h, d, e

After the edges are traversed in this way, all statements have been executed once. Other combinations of edges of the graph can also be used to achieve complete coverage. But the cost of testing should always be minimized, which means reaching the goal with the smallest possible number of test cases.

The expected results and the expected behavior of the test object should be identified in advance from the specification (not the code!). After execution, the expected and actual results, and the behavior of the test object, must be compared to detect any difference or failure.

*Coverage of the nodes of the control flow*

*One test case is enough*

### Definition of the Test Exit Criteria

The exit criteria for the tests can be very clearly defined:

→Statement coverage =

$$(\text{number of executed statements} / \text{total number of statements}) \times 100\%$$

*C0-measure* Statement coverage is also known as C0-coverage (C-zero). It is a very weak criterion. However, sometimes 100% statement coverage is difficult to achieve, as when, for instance, exception conditions appear in the program that can be triggered only with great trouble or not at all during test execution.

### The Value of the Technique

*Unreachable code can be detected*

If complete coverage of all statements is required and some statements cannot be executed by any test case, this may be an indication of unreachable source code (dead statements).

*Empty ELSE parts are not considered*

If a condition statement (IF) has statements only after it is fulfilled (i.e., after the THEN clause) and there is no ELSE clause, then the control flow graph has a THEN edge, starting at the condition, with (at least) one node, but additionally a second outgoing ELSE edge without any intermediate nodes. The control flow of both of these edges is reunited at the terminating (ENDIF) node. For statement coverage, an empty ELSE edge (between IF and ENDIF) is irrelevant. Possible missing statements in this program part are not detected by a test using this criterion!

Statement coverage is measured using test tools (section 7.1.4).

## 5.2.2 Decision/Branch Testing and Coverage

A more advanced criterion for white box testing is →branch coverage of the control flow graph; for example, the edges (connections) in the graph are the center of attention. This time, the execution of decisions is considered instead of the execution of the statement. The result of the decision determines which statement is executed next. This should be used in testing.

→branch or decision test

If the basis for a test is the control flow graph with its nodes and edges, then it is called a branch test or branch coverage. A branch is the connection between two nodes of the graph. In the program text, there are IF or CASE statements, loops, and so on, also called *decisions*. This test is thus called *decision test* or *decision coverage*. There may be differences in the

degree of coverage. The following example illustrates this: An IF statement with an empty ELSE-part is checked. Decision testing gives 50% coverage if the condition is evaluated to true. With one more test case where the condition is false, 100% decision coverage will be achieved. For the branch test, which is built from the control flow graph, slightly different values result. The THEN part consists of two branches and one node, the ELSE part only of one branch without any node (no statement there). Thus, the whole IF statement with the empty ELSE part consists of three branches. Executing the condition with true results in covering two of the three branches, that is, 66% coverage. (Decision testing gives 50% in this case). Executing the second test case with the condition being false, 100% branch coverage and 100% decision coverage are achieved. Branch testing is discussed further a bit later.

Thus, contrary to statement coverage, for branch coverage it is not interesting if, for instance, an IF statement has no ELSE-part. It must be executed anyway. Branch coverage requires the test of every decision outcome: both THEN and ELSE in the IF statement; all possibilities for the CASE statement and the fall-through case; for loops, both execution of the loop body, bypassing the loop body and returning to the start of the loop.

*Empty ELSE-parts are considered*

### Test Cases

In the example (figure 5-9), additional test cases are necessary if all branches of the control flow graph must be executed during the test. For 100% statement coverage, a test case executing the following order of edges was sufficient:

a, b, f, g, h, d, e

The edges c, i, and k have not been executed in this test case. The edges c and k are empty branches of a condition, while the edge i is the return to the beginning of the loop. Three test cases are necessary:

a, b, c, d, e

a, b, f, g, i, g, h, d, e

a, k, e

All three test cases result in complete coverage of the edges of the control flow graph. With that, all possible branches of the control flow in the source code of the test object have been tested.

*Additional test cases necessary*

*Connection (edge) coverage of the control flow graph*

Some edges have been executed more than once. This seems to be redundant, but it cannot always be avoided. In the example, the edges a and e are executed in every test case because there is no alternative to these edges.

For each test case, in addition to the preconditions and postconditions, the expected result and expected behavior must be determined and then compared to the actual result and behavior. Furthermore, it is reasonable to record which branches have been executed in which test case in order to find wrong execution flows. This helps to find faults, especially missing code in empty branches.

### Definition of the Test Exit Criteria

As with statement coverage, the degree of branch coverage is defined as follows:

$$\text{Branch coverage} = \left( \frac{\text{number of executed branches}}{\text{total number of branches}} \right) \times 100\%$$

*C1-measure* Branch coverage is also called C1-coverage. The calculation counts only if a branch has been executed at all. The frequency of execution is not relevant. In our example, the edges a and e are each passed three times—once for each test case.

If we execute only the first three test cases in our example (not the fourth one), edge k will not be executed. This gives a branch coverage of 9 executed branches out of 10 total:

$$(9 / 10) \times 100\% = 90\%.$$

For comparison, 100% statement coverage has already been achieved after the first test case.

Depending on the criticality of the test object, and depending on the expected failure risk, the test exit criterion can be defined differently. For instance, 85% branch coverage can be sufficient for a component of one project, whereas for a different project, another component must be tested with 100% coverage. The example shows that the test cost is higher for higher coverage requirements.

### The Value of the Technique

*More test cases necessary*

Decision/branch coverage usually requires the execution of more test cases than statement coverage. How much more depends on the structure of the

test object. In contrast to statement coverage, branch coverage makes it possible to detect missing statements in empty branches. Branch coverage of 100% guarantees 100% statement coverage, but not vice versa. Thus, branch coverage is a stronger criterion.

Each of the branches is regarded separately and no particular combinations of single branches are required.

- 
- A branch coverage of 100% should be aimed for.
  - The test can only be categorized as sufficient if, in addition to all statements, every possible branch of the control flow, and thus every possible result of a decision in the program text, is considered during test execution.
- 

**Hint**

For object-oriented systems, statement coverage as well as branch coverage are inadequate because the control flow of the functions in the classes is usually short and not very complex. Thus, the required coverage criteria can be achieved with little effort. The complexity in object-oriented systems lies mostly in the relationship between the classes, so additional adequate coverage criteria are necessary in this case. As tools often support the process of determining coverage, coverage data can be used to detect not-called methods or program parts.

*Inadequate for object-oriented systems*

### 5.2.3 Test of Conditions<sup>27</sup>

Branch coverage exclusively considers the logical value of the result of a condition (“true” or “false”). Using this value, it is decided which branch in the control flow graph to choose and, accordingly, which statement is executed next in the program. If a decision is based on several (partial) conditions connected by logical operators, then the complexity of the condition should be considered in the test. The following sections describe different requirements and degrees of test intensity under consideration of combined conditions.

*Considering the complexity of combined conditions*

---

27. In the ISTQB Certified Tester syllabus, condition test and multiple condition testing are mentioned only as examples of further structure-based techniques. These two techniques, as well as condition determination testing, are described here anyway because they are effective techniques for testing conditions.

### Condition Testing and Coverage

The goal of condition testing is to cause each  $\rightarrow$ atomic (partial) condition in the test to adopt both a *true* and a *false* value.

*Definition of an atomic partial condition*

An atomic partial condition is a condition that has no logical operators such as AND, OR, and NOT but at the most includes relation symbols such as  $>$  and  $=$ . A condition in the source code of the test object can consist of multiple atomic partial conditions.

#### **Example for combined conditions**

An example for a combined condition is  $x > 3$  OR  $y < 5$ . The condition consists of two conditions ( $x > 3$ ;  $y < 5$ ) connected by the logical operator OR.

The goal of condition testing is that each partial condition (i.e., each individual part of a combined condition) is evaluated once, resulting in each of the logical values. The test data  $x = 6$  and  $y = 8$  result in the logical value *true* for the first condition ( $x > 3$ ) and the logical value *false* for the second condition ( $y < 5$ ). The logical value of the complete condition is *true* (*true* OR *false* = *true*). The second pair of test data with the values  $x = 2$  and  $y = 3$  results in *false* for the first condition and *true* for the second condition. The value of the complete condition results in *true* again (*false* OR *true* = *true*). Both parts of the combined condition have each resulted in both logical values. The result of the complete condition, however, is equal for both combinations.

*A weak criterion*

Condition coverage is therefore a weaker criterion than statement or branch coverage because it is not required that different logical values for the result of the complete condition are included in the test.

### Multiple Condition Testing and Coverage

*All combinations of the logical values*

Multiple condition testing requires that all *true-false* combinations of the atomic partial conditions be exercised at least once. All variations should be built, if possible.

#### **Continuation of the example**

Four combinations of test cases are possible with the test data from the previous example for the two conditions ( $x > 3$ ,  $y < 5$ ):

$x = 6$  (T),  $y = 3$  (T),  $x > 3$  OR  $y < 5$  (T)

$x = 6$  (T),  $y = 8$  (F),  $x > 3$  OR  $y < 5$  (T)

$x = 2$  (F),  $y = 3$  (T),  $x > 3$  OR  $y < 5$  (T)

$x = 2$  (F),  $y = 8$  (F),  $x > 3$  OR  $y < 5$  (F)

*Multiple condition testing subsumes statement and branch coverage*

The evaluation of the complete condition results in both logical values. Thus, multiple condition testing meets the criteria of statement and branch coverage. It is a more comprehensive criterion that also takes into account the complexity of combined conditions. But this is a very expensive technique due to the growing

number of atomic partial conditions that make the number of possible combinations grow exponentially (to  $2^n$ , with  $n$  being the number of atomic partial conditions).

A problem results from the fact that test data cannot always generate all combinations.

*Not all combinations are always possible*

An example should clarify this. For the combined condition of  $3 \leq x$  AND  $x < 5$  not all combinations with the according values for the variable  $x$  can be produced because the parts of the combined condition depend on each other:

$x = 4$ :  $3 \leq x$  (T),  $x < 5$  (T),  $3 \leq x$  AND  $x < 5$  (T)

$x = 8$ :  $3 \leq x$  (T),  $x < 5$  (F),  $3 \leq x$  AND  $x < 5$  (F)

$x = 1$ :  $3 \leq x$  (F),  $x < 5$  (T),  $3 \leq x$  AND  $x < 5$  (F)

$x = ?$ :  $3 \leq x$  (F),  $x < 5$  (F), combination not possible because the value  $x$  shall be smaller than 3 and greater than or equal to 5 at the same time.

**Example for not feasible combinations of partial condition**

### Condition Determination Testing / Minimal Multiple Condition Testing

Condition determination testing eliminates the problems discussed previously. Not all combinations must be included; however, include every possible combination of logical values where the modification of the logical value of an atomic partial condition can change the logical value of the whole combined condition. Stated in another way, for a test case, every atomic partial condition must have a meaningful impact on the result. Test cases in which the result does not depend on a change of an atomic partial condition need not be designed.

*Restriction of the combinations*

For clarification, we revisit the example with the two atomic partial conditions ( $x > 3$ ,  $y < 5$ ) and the OR-connection ( $x > 3$  OR  $y < 5$ ). Four combinations are possible ( $2^2$ ):

1)  $x = 6$  (T),  $y = 3$  (T),  $x > 3$  OR  $y < 5$  (T)

2)  $x = 6$  (T),  $y = 8$  (F),  $x > 3$  OR  $y < 5$  (T)

3)  $x = 2$  (F),  $y = 3$  (T),  $x > 3$  OR  $y < 5$  (T)

4)  $x = 2$  (F),  $y = 8$  (F),  $x > 3$  OR  $y < 5$  (F)

For the first combination, the following applies: If the logical value is wrongly calculated for the first condition (i.e., an incorrect condition is implemented), then the fault can change the logical value of the first condition part from *true* (T) to *false* (F). But the result of the complete condition stays unchanged (T). The same applies for the second partial condition.

**Continuation of the example**

*Changing a partial condition without changing the result*

For the first combination, incorrect results of each partial condition are masked because they have no effect on the result of the complete condition and thus failures will not become visible at the outside. Consequently, the test with the first combination can be left out.

If the logical value of the first partial condition in the second test case is calculated wrongly as *false*, then the result value of the combined condition changes from *true* (T) to *false* (F). A failure then becomes visible because the value of the combined condition has also changed. The same applies for the second partial condition in the third test case. In the fourth test case, an incorrect implementation is detected as well because the logical value of the complete condition changes.

---

#### *Small number of test cases*

For every logical combination of the combined decision, it must be decided which test cases are sensitive to faults and for which combinations faults can be masked. Combinations where faults are masked need not be considered in the test. Here, the number of test cases is significantly smaller than in multiple condition testing.

#### **Test Cases**

For designing the test cases, it must be considered which input data leads to which result of the decisions or partial conditions and which parts of the program will be executed after the decision. The expected output and expected behavior of the test object should also be defined in advance in order to detect whether the program behaves correctly.

#### **Hint**

- Because of the weak significance, condition testing should not be used.
  - For complex conditions, condition determination testing should be applied because the complexity of the conditional expression is taken into account for test case design. The method also subsumes statement and branch coverage, which means they need not be checked in addition.
- 

However, it may be very expensive to choose the input values in such a way that a certain part of the condition gets the logical value required by the test case.

#### **Definition of the Test Exit Criteria**

Analogous to the previous techniques, the proportion between the executed and all the required logical values of the (partial) condition (parts) can be calculated. For the techniques, which concentrate on the complexity

of the decisions in the source code, it is reasonable to try to achieve a complete verification (100% coverage). If complexity of the decisions is not important in testing, branch coverage can be seen as sufficient.

### The Value of the Technique

If complex decisions are present in the source code, they must be tested intensively to detect possible failures. Combinations of logical expressions are especially defect prone. Thus, a comprehensive test is very important. However, condition determination testing is an expensive technique for test case design.

*Complex conditions are often defect prone*

- 
- It can be reasonable to split combined complex conditions into a tree structure of nested simple conditions and then execute a branch coverage test for these sequences of conditions.
  - The intensive test of complex conditions can possibly be omitted if they have been subjected to a review (section 4.1.2) in which the correctness is verified.
- 

**Hint**

A disadvantage of condition testing is that it checks Boolean expressions only inside a statement (for example, IF statement). In the following example of a program fragment, the following fact remains undetected: the IF condition actually consists of multiple parts and condition determination testing needs to be applied.

**Excursion**

```
...
Flag = (A || (B && C));
If (Flag)
    ...;
else ...;
...
```

This particular disadvantage can be circumvented if all Boolean expressions that occur are used as a basis for the creation of test cases.

Another problem occurs in connection with measuring the coverage of (partial) conditions. Some compilers shortcut the evaluation of the Boolean expression as soon as the total result of the decision is known. For instance, if the value FALSE has been detected for one of two condition parts of an AND-combination, then the complete condition is FALSE regardless of the result of the second condition part. Some compilers even change the order of the evaluation, depending on the Boolean operators, to get the final result as quickly as possible and to be able to disregard any other partial conditions. Test cases that are supposed to achieve 100% coverage can be executed, but because of the shortened evaluation, this coverage cannot be verified.

*The compiler terminates evaluation of expressions*

**Excursion: Path Testing and Coverage<sup>28</sup>**

*All possible paths through a test object*

Until now, test case determination focused on the statements or branches of the control flow as well as the complexity of decisions. If the test object includes loops or repetitions, the previous considerations are not sufficient for an adequate test. Path coverage requires the execution of all different paths through the test object.

**Example for a path test**

To clarify the use of the term *path*, consider the control flow graph in figure 5-9.

The program fragment represented by the graph contains a loop. This DO-WHILE loop is executed at least once. In the WHILE condition at the end of the loop, it is decided whether the loop must be repeated, that is, if a jump back to the start of the loop is necessary. When using branch coverage for test design, the loop has been considered in two test cases:

- Loop without repetition:  
a, b, f, g, h, d, e
- Loop with single return (i) and a single repetition:  
a, b, f, g, i, g, h, d, e

Usually a loop is repeated more than once. Further possible sequences of branches through the graph of the program are as follows:

- a, b, f, g, i, g, i, g, h, d, e
- a, b, f, g, i, g, i, g, i, g, h, d, e
- a, b, f, g, i, g, i, g, i, g, i, g, h, d, e
- etc.

This shows that there are an indefinite number of paths in the control flow graph. Even with restrictions on the number of loop repetitions, the number of paths increases indefinitely (see also section 2.1.4).

*Combination of program parts*

A path describes the possible order of single program parts in a program fragment.

Contrary to this, branches are viewed independently, each for itself. The paths consider dependencies between the branches, as for example with loops, at which one branch leads back to the beginning of another branch.

**Example: Statement and branch coverage in VSR**

In section 5.1.1 for the function `calculate_price()` of the VSR subsystem *DreamCar*, test cases have been derived from valid and invalid equivalence classes of the parameters. In the following code, test cases are evaluated by their

28. Path testing is not mentioned in the ISTQB Certified Tester Foundation Level syllabus. It is described here because it can be seen as a “further step” in statement and branch coverage and because the term is often misunderstood.

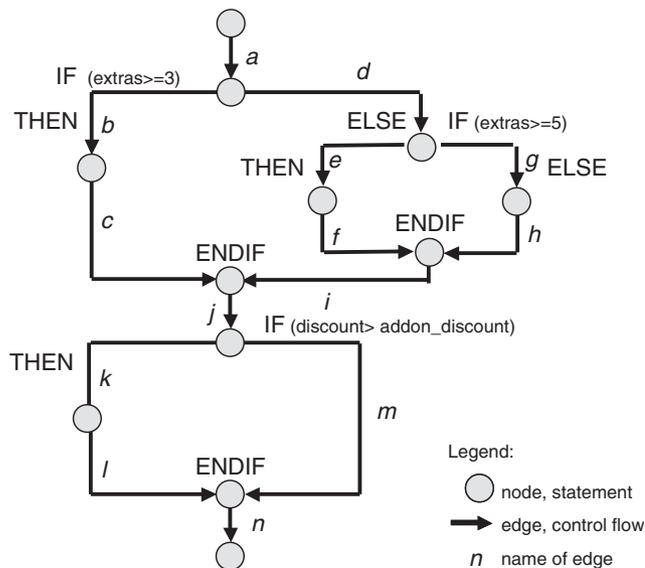
ability to cover the source code, that is, execute respective parts of the method. Branch coverage of 100% should be achieved to ensure that during test execution all branches have been executed at least once.

For better understanding, the source code of the function from section 3.2.3 is repeated here:

```
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);
}
```

The control flow graph of the function `calculate_price()` is shown in figure 5-10.



**Figure 5-10**

Control flow graph of the function `calculate_price()`

In section 3.2.3, the following two test cases have been chosen:

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

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

The test cases cause the execution of the following edges of the graph:

Test case 01: a, b, c, j, m, n  
 Test case 02: a, b, c, j, m, n

*43% branch coverage  
 achieved*

The edges d, e, f, g, h, i, k, l have not been executed. The two test cases covered only 43% of the branches (6 out of 14). Test case 02 gives no improvement of the coverage and is not necessary for branch coverage. However, considering the specification, test case 02 should have led to execution of more statements because a different discount should have been calculated (with five or more pieces of extra equipment).

To increase test coverage, the following further test cases are specified:

```
// testcase 03
price = calculate_price(10000.00,2000.00,1000.00,0,10);
test_ok = test_ok && (abs(price-12000.00) < 0.01);

// testcase 04
price = calculate_price(25500.00,3450.00,6000.00,6,15);
test_ok = test_ok && (abs(price-30225.00) < 0.01);
```

These test cases cause the execution of the following edges of the graph:

Test case 03: a, d, g, h, i, j, k, l, n  
 Test case 04: a, b, c, j, k, l, n

*86% path coverage achieved*

These test cases lead to execution of further edges (d, g, h, i, k, and l) and thus increase branch coverage to 86%. Edges e and f have not yet been executed.

*Evaluation of the conditions*

Before trying to reach the missing edges by further test cases, the conditions of the IF statements are analyzed more closely, that is, the source code is analyzed in order to define further test cases. To get to the edges e and f, the result of the first condition ( $\text{extras} \geq 3$ ) must be false in order to execute the ELSE-part. In this ELSE-part, the condition ( $\text{extras} \geq 5$ ) must be true. Therefore, a value has to be found that meets the following condition:

$$\neg(\text{extras} \geq 3) \text{ AND } (\text{extras} \geq 5)$$

No such value exists and the missing edges can never be reached. The source code contains a defect.

---

This example should clarify the relationship between statement, branch, and path coverage as well. The test object consists of altogether three IF statements; two are nested and the third is placed separately from the others (figure 5-10).

All statements (nodes) are executed by the following sequence of edges in the graph:

a, b, c, j, k, l, n  
a, d, e, f, i, j, k, l, n  
a, d, g, h, i, j, k, l, n

These sequences are sufficient to achieve 100% statement coverage. But not all branches (edges) have been covered yet. The edge m is still missing. An execution sequence might look like this:

a, b, c, j, m, n

This new sequence can replace the first execution sequence shown previously. With the resulting three test cases, which result in these three execution sequences, 100% branch coverage is achieved.

But, even for this simple program fragment, there are still possibilities to traverse the graph differently and thus take care of all paths through the graph. Until now, the following paths have not been executed yet:

a, d, e, f, i, j, m, n  
a, d, g, h, i, j, m, n

Altogether, six different paths through the source code result (the three possible paths through the graph before edge j multiplied by two for the two possible paths after edge j). There is the precondition that the conditions are independent from each other and the edges can be combined freely.

**Example:**  
**Relationship between  
the measures**

*Further paths  
through the graph*

If there are loops in the source code, then every possible number of loop repetitions is counted as a possible path through the program fragment. It is obvious that 100% path coverage in testing is not feasible for a nontrivial program.

## 5.2.4 Further White Box Techniques

In addition to the most common techniques described here, there are a number of other white box test techniques that can be used for evaluating test objects. You can find out more about them in [Myers 79], [Beizer 90], and [Pol 98]. The following section describes one technique in a little more detail.

**Excursion***Data-flow-based techniques*

A number of techniques use the flow of data through the test object as the basis for identifying test cases. Primarily, the data usages in the test object are verified. The use of each variable is analyzed, whereby the definitions of and read/write access to variables are distinguished from each other. These techniques may find faults where a value of a variable in the test object causes failures when it is used in other places. Furthermore, the technique verifies whether the value of a variable is used to create other variables or if it is used to calculate the logical value of a condition. This information allows defining various data flow criteria, which can then be covered by test cases. A detailed description of data-flow-based techniques can be found in [Clarke et al. 85].

**5.2.5 General Discussion of the White Box Technique***Determine the test intensity*

The basis for all of the white box techniques described is the source code.

Adequate test case design techniques can be chosen and applied depending on the complexity of the program structure. The intensity of the test depends on the source code and the selected technique.

*Useful for lower test levels*

The white box techniques we've described are appropriate for the lower test levels. For example, it is not very reasonable to require coverage of single statements or branches in a system test because system testing is not the right method to check single statements or conditions in the code.

*Coverage is desirable even at higher test levels*

The concept of coverage can be applied to other test levels above the code level. For example, during an integration test, we can assess what percentage of modules, components, or classes are executed during the test. This results in module, component, or class coverage. The required percentage value can be determined in advance and checked during test execution.

*"Missing source code" is not considered*

Missing implementation of requirements is impossible to find for white box techniques. White box techniques can verify only code that exists, that is, requirements that are implemented in the program, not code that should be there but isn't. Thus, other test design techniques are required to find omissions.

**5.2.6 Instrumentation and Tool Support***Determination of the executed program parts*

Code-based white box techniques require that different program parts are executed and that conditions get different logical values. To be able to evaluate the test, it must be determined which program parts have already been executed and which haven't. To do this, the test object must be instrumented at strategically relevant spots before test execution. →Instrumentation often works this way: The tool inserts counters in the

program and initializes them with zero. During program execution, the counters are incremented when they are passed. At the end of the test execution, the counters contain the number of passes through the corresponding program parts. If a counter remained zero during the test, then the corresponding program part has not been executed.

The instrumentation, the evaluation of the test runs, and the calculation of the achieved coverage should not be done manually because this would require too many resources and a manual instrumentation is error prone. Numerous tools perform these tasks (see section 7.1.4). These tools are important for white box testing because they increase productivity and indirectly improve the quality of the test object.

*Use tools*

### 5.3 Intuitive and Experience-Based Test Case Determination

Besides the systematic approaches, intuitive determination of test cases should be performed. The systematically identified test cases may be supplemented by test cases designed using the testers' intuition. The techniques are also called experience based because they depend on the experience of the testers. Intuitive testing can detect faults overlooked by systematic testing. It is therefore always advisable to perform additional intuitive testing.

The basis of this method is the skill, experience, and knowledge of the tester to select test cases that uncover expected problems and their symptoms (failures). A systematic approach is not used. The test cases are based on where faults have occurred in the past or the tester's ideas of where faults might occur in the future. This type of test case design is also called *error guessing* and is used very often in practice.

*Intuitive skill and experience of the testers*

Knowledge in developing similar applications and using similar technologies should also be used when designing test cases, in addition to experience in testing. If, for example, failures were found in previous projects in which a certain programming language was used, it is reasonable to use those failures when you are designing the tests in the actual project if you are using the programming language that caused the failures. One technique for intuitive testing, exploratory testing, will be discussed in more detail.

If the documents, which form the basis for test design, are of low quality, are obsolete, or do not exist at all, so-called *exploratory testing* may

*Exploratory testing*

help. In the extreme case, only the program exists. The technique is also applicable when time is severely restricted because it uses much less time than other techniques. The approach is mainly based on the intuition and experience of the tester.

*The approach  
of exploratory testing*

The test activities in exploratory testing are executed nearly in parallel. The structured test process is not applied. An explicit previous planning of the test activities is not done. The possible elements of the test object (its specific tasks and functions) are “explored.” It is then decided which parts will be tested. Only a few test cases are executed and their results are analyzed. By executing the test cases, the “unknown” behavior of the test object will be determined further. Anything considered “interesting,” as well as other information, is then used to determine the next test cases. In this step-by-step manner, knowledge about the test object under test is collected. It becomes clearer what the test object does and how it works, which quality problems there could be, and which expectations to the program should be fulfilled. One result of exploratory testing may be that it becomes clear which test techniques can be applied if there is time left in the project.

*Test charter*

It makes sense to restrict exploratory testing to certain elements of the program (certain tasks or functions). The elements are further decomposed. The term *test charter* is used for such smaller parts. It should not take more than one or two hours to test a test charter. When executing test charters, the following questions are of interest:

- Why? What is the goal of the test run?
- What? What is to be tested?
- How? Which testing method should be used?
- What? What kind of problems should be found?

*Main features  
of exploratory testing*

The generic ideas of exploratory testing are as follows:

- Results of one test case influence the design and execution of further test cases.
- During testing, a “mental” model of the program under test is created. The model contains how the program “works” and how it behaves or how it should behave.
- The test is run against this model. The focus is to find further aspects and behaviors of the program that are still not part of the “mental” model or are differing from aspects found before.

All the approaches for intuitive test case determination cannot be associated explicitly with white box or black box techniques because neither the requirements nor the source code are exclusively the basis for the considerations and tests. They should be applied more in the higher test levels. In the lower ones, usually sufficient information such as source code or detailed specification is accessible for applying systematic techniques.

*Neither black box  
nor white box*

Intuitive test case determination should not be applied as the primary testing technique. Instead, this technique should be used complete the test cases and to support the systematic test design techniques.

*Not to be used as first or only  
technique*

### Test Cases

Knowledge for determination of additional test cases can be drawn from many sources.

---

In the development project for the *CarConfigurator*, the testers are very familiar with the previous system. Many of them have tested this system before. They know the weaknesses the system had and they know the problems the car dealers had with the operation of the old software (from hotline data and from discussions with car salespeople). Employees from the marketing department know for the business-related test which vehicles in which configurations are sold often and which theoretically possible combinations of extra equipment might not even be shippable. They use this experience to intuitively prioritize the systematically identified test cases and to complete them by additional test cases. The test manager knows which of the developer teams act under the most severe time pressure and even work on weekends. Hence, she will test the components from these teams more intensively.

---

**Example:**  
**Tester knowledge for  
the CarConfigurator**

Testers should use all their knowledge to find additional test cases. Naturally, the pre- and postconditions, the expected outcomes, and the expected behavior of the test object must be defined in advance for intuitive testing as well.

*Using all knowledge*

- 
- Because extensive experience is often only available in the minds of the experienced testers, maintaining a list with possible errors, faults, and suspicious situations might be very helpful. Frequently occurring errors, faults, and failures are noted in the list and are thus available to all the testers. With the help of the possible trouble areas and critical situations that have been identified, additional test cases can be designed.

**Hint**

- The list may even be beneficial to developers because it indicates in advance what potential problems and difficulties might occur. These can be considered during implementation and thus serve for error prevention.

---

### Definition of the Test Exit Criteria

*A test exit criterion is not definable*

Unlike with the systematic techniques, a criterion for termination cannot be specified. If the previously mentioned list exists, then a certain completeness can be verified against the list.

### The Value of the Technique

*Mostly successful in finding more defects*

Intuitive test case determination and exploratory testing can often be used with good success. They are a sensible addition to systematic techniques. The success and effectiveness of this approach depend very much on testers' skill and intuition and their previous experience with similar applications and the technologies used. Such approaches can also contribute to finding holes and errors in the risk analysis. If intuitive testing is applied in addition to systematic testing, inconsistencies in the test specification not previously detected can be found. Intensity and completeness of intuitive and exploratory test design cannot be measured.

## 5.4 Summary

*Which technique and when to use it*

This chapter has introduced a number of *techniques* for testing of software.<sup>29</sup> The question is, When should each technique be applied? The following list includes answers to this question and presents a reasonable approach. The general goal is to identify sufficiently different test cases in order to be able to find existing faults with a certain probability and with as little effort as possible. The techniques for test design should therefore be chosen appropriately.

However, before designing a test, you should check some factors that have considerable influence on the selection or even prescribe the application of certain test methods. The selection of techniques can depend on the following different circumstances and conditions:

---

29. There exist other techniques not described in this book. This applies especially to integration testing, testing of distributed applications, and testing of real-time and embedded systems. Some of these techniques are part of the Advanced Level Tester certification scheme. More information can be found in [Bath 08].

---

- **The kind of test object**

The complexity of the program text can vary considerably. Depending on this, adequate test techniques should be chosen. If, for example, decisions in the program are combined from several atomic conditions, branch coverage is not sufficient. A suitable technique to check the conditions should be chosen depending on the criticality and the risk in case of failure.

- **Formal documentation and the availability of tools**

If specification or model information is available in a formal document, it can be fed directly into test design tools, which then derive test cases. This very much decreases the effort required to design the tests.

- **Conformance to standards**

Industry and regulatory standards may require use of certain test techniques and coverage criteria. Compliance to such standards is often mandatory for safety-critical software or when high reliability is required.

- **Tester experience**

Tester experience may lead to the choice of special techniques. A tester will, for example, reuse techniques that have led to finding serious faults earlier.

- **Customer wishes**

The customer may require the use of specific test techniques and even the test coverage to be achieved. This has the advantage that at least these techniques will be applied during development. This may lead to fewer failures in acceptance testing.

- **Risk assessment**

The expected risk dictates more or less the test activities, that is, the choice of techniques and the intensity of the execution. Risk-prone areas should be tested more thoroughly.

- **Additional factors**

Finally, there are factors like the availability of the specification and other documentation, the knowledge and skill of the test personnel, time and budget, the test level, and previous experience with what kind of defects occur most often and with which test techniques these have been found. They can all have a large influence on selecting the testing techniques.

Test design techniques cannot be specified in a standard way. Their selection should always be based on a thoughtful decision. The following list should help in selecting the most applicable test technique.

- Testing functionality* ■ The system functioning correctly is certainly of great relevance. The functionality of the test object must be sufficiently verified. The development of all test cases, regardless of which technique or procedure was used, includes determining the expected results and reactions of the test object. This ensures a verification of the functionality for every evaluation of the test cases. Comparing the actual with the expected outputs and reactions contains a verification of the functionality.
- Equivalence class partition combined with boundary value analysis* ■ When you are designing test cases, equivalence class partitioning in combination with boundary value analysis should be applied for every test object. When you are executing these test cases, you should use the appropriate tools for measuring code coverage to find the test coverage already achieved (see section 7.1.4).
- Consider execution history* ■ If different states have an influence on the operating sequence in the test object, state transition testing must be applied. Only state transition testing verifies the cooperation of the states, transitions, and the corresponding behavior of the functions adequately.
  - If dependencies between the input data are given and must be taken care of in the test, these dependencies can be modeled using cause-effect graphs or decision tables. The corresponding test cases can be taken from the decision table.
  - If there are many parameters or settings with unknown dependencies, use pairwise testing for them.
  - For system testing, use cases (displayed in use case diagrams) can be applied as a basis for designing test cases.
  - In component and integration testing, coverage measurements should be included with all black box techniques. The parts of the test object still not executed should then be specifically tested with a white box test design technique. Depending on the criticality and nature of the test object, appropriate extensive white box technique must be selected.
- Minimum criterion: branch coverage* ■ The minimum criterion should be 100% branch coverage. If complex decisions exist in the test object, then condition determination testing is the appropriate technique to find faulty decisions.
  - In measuring coverage, loops should be repeated more than once. For critical parts of the system, the loops must be verified using the appropriate methods (boundary interior-path test and structured path test [Howden 75]).

- 
- Complete path coverage of a test object is usually not achievable. It should be considered a mere theoretical measure and is of little practical importance because of the great cost and because it is impossible to achieve for programs with loops. It is very seldom used in practice.
  - It is sensible to apply white box techniques at lower test levels while black box techniques can be applied in all test levels, especially the higher ones.
  - Intuitive determination of test cases should not be ignored. It is a good supplement to systematic test design methods.

- 
- Testing always comprises the combination of different techniques because no testing technique exists that covers all aspects to be considered in testing equally well.
  - The criticality and the expected risk in case of failure guide the selection of the testing techniques and the intensity of the execution.
  - The basis for the selection of the white box technique is the structure of the test object. If, for example, no complex decisions exist in the test object, the use of condition determination testing makes no sense.
- 

**Hint**



## 6 Test Management

*This chapter describes ways to organize test teams, which qualifications are important, the tasks of a test manager, and which supporting processes must be present for efficient testing.*

### 6.1 Test Organization

#### 6.1.1 Test Teams

Testing activities are necessary during the entire software product life cycle (see chapter 3). They should be well coordinated with the development activities. The easiest solution is to let the developer perform the testing.

However, because there is a tendency to be blind to our own errors, it is much more efficient to let different people perform testing and development and to organize testing as independently as possible from development.

Independent testing provides the following benefits:

- Independent testers are unbiased and thus find additional and different defects than developers find.
- An independent tester can verify (implicit) assumptions made by developers during specification and implementation of the system.

*Benefits  
of independent testing*

But there may also be drawbacks to independent testing:

- Too much isolation may impair the necessary communication between testers and developers.
- Independent testing may become a bottleneck if there is a lack of necessary resources.
- Developers may lose a sense of responsibility for quality because they may think, “the testers will find the →problems anyway.”

*Possible drawbacks of  
independent testing*

*Models of independent testing*

The following models or options for independence are possible:

1. The development team is responsible for testing, but developers test each other's programs, i.e., a developer tests the program of a colleague.
2. There are testers within the development team; these testers do all the test work for their team.
3. One or more dedicated testing teams exist within the project team (these teams are not responsible for development tasks). Typically, team members from the business or IT department work as independent testers.
4. Independent test specialists are used for specific testing tasks (such as performance test, usability test, security test, or for showing conformance to standards and regulatory rules).
5. A separate organization (testing department, external testing facility, test laboratory) takes over the testing (or important parts of it, such as the system test).

*When to choose which model*

For each of these models, it is advantageous to have testing consultants available. These consultants can support several projects and can offer methodical assistance in areas such as training, coaching, test automation, etc. Which of the previously mentioned models is appropriate depends on—among other things—the current test level.

**Component Testing**

Testing should be close to development. Although often used, it is definitely the worst choice to allow developers to test their own programs. Independent testing such as in model 1 is easy to organize and would certainly improve quality. Testing such as in model 2 is useful, if a sufficient number of testers relative to the number of developers can be made available. However, with both testing models, there is the risk that the participating people essentially consider themselves developers and thus will neglect their testing responsibilities.

To prevent this, the following measures are recommended:

**Hint**

- Project or test management should set testing standards and rules, and require test logs from the developers.
- To provide support for applying systematic testing methods, testing specialists should, at least temporarily, be called in as coaches.

## Integration Testing

When the same team that developed the components also performs integration and integration testing, this testing can be organized as for component testing (models 1, 2).

If components originating from several teams are integrated, then either a mixed integration team with representatives from the involved development groups or an independent integration team should be responsible. The individual development team may have their own view about their own component and therefore may overlook faults. Depending on the size of the development project and the number of components, models 3, 4, and 5 should be considered here.

## System Testing

The final product shall be considered from the point of view of the customer and the end user. Therefore, independence from the development team is crucial. This leaves only models 3, 4, and 5.

---

In the VSR project, each development team is responsible for component testing. These teams are organized according to the previously mentioned models 1 and 2. In parallel to these development teams, an independent testing group is established. This testing group is responsible for integration and system testing. Figure 6-1 shows the organization.

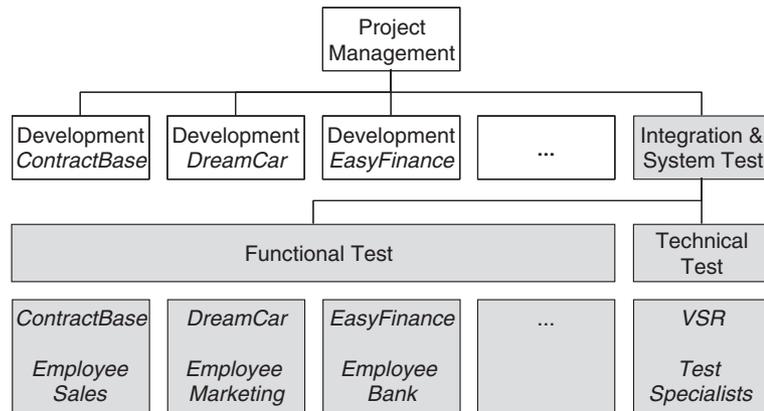
Two or three employees from each responsible user department (sales, marketing, etc.) are made available for the functional or business-process-based testing of every subsystem (*ContractBase*, *DreamCar*, etc.). These people are familiar with the business processes to be supported by the particular subsystem and know the requirements “their” test object should fulfill from the users’ point of view. They are experienced PC users, but not IT experts. It is their task to support the test specialists in specifying business-related functional test cases and to perform these tests. When the testing activities are started, they have received training in basic testing procedures (test process, specification, execution, and logging).

Additionally, test personnel consists of three to five IT and test specialists, responsible for integration activities, nonfunctional tests, test automation, and support of test tools (“technical test”). A test manager, responsible for test planning and test control, is in charge of the test team. The manager’s tasks also comprise coaching of the test personnel, especially instructing the staff on testing the business requirements.

**Example:**  
**Organization of the**  
**VSR tests**

→ *Test logging*

**Figure 6-1**  
VSR project organization



### 6.1.2 Tasks and Qualifications

Specialists with knowledge covering the full scope of activities in the test process should be available. The following roles should be assigned, ideally to specifically qualified employees:

*Roles and qualification profiles*

- **Test manager** (test leader): Test planning and test control expert(s), possessing knowledge and experience in the fields of software testing, quality management, project management, and personnel management. Typical tasks may include the following:
  - Writing and coordinating the test policy for the organization
  - Developing the test approach and test plan as described in section 6.2.2
  - Representing the testing perspective in the project
  - Procuring testing resources
  - Selecting and introducing suitable test strategies and methods, introducing or improving testing tools, organizing tools training, deciding about test environment and test automation
  - Introducing or optimizing supporting processes (e.g., problem management, configuration management) in order to be able to trace back changes and securing reproducibility of the tests
  - Introducing, using, and evaluating metrics defined in the test plan
  - Regularly adapting test plans based on test results and test progress
  - Identifying suitable metrics for measuring test progress, and evaluating the quality of the testing and the product
  - Writing and communicating test reports

- **Test designer** (test analyst): Expert(s) in test methods and test specification, having knowledge and experience in the fields of software testing, software engineering, and (formal) specification methods. Typical tasks may include the following:
  - Reviewing requirements, specifications, and models for testability and in order to design test cases
  - Creating test specifications
  - Preparing and acquiring test data
- **Test automator:** Test automation expert(s) with knowledge of testing basics, programming experience, and deep knowledge of the testing tools and script languages. Automates tests as required, making use of the test tools available for the project.
- **Test administrator:** Expert(s) for installing and operating the test environment (system administrator knowledge). Sets up and supports the test environment (often coordinating with general system administration and network management).
- **Tester:**<sup>1</sup> Expert(s) for executing tests and reporting failures (IT basics, basic knowledge of testing, using the test tools, understanding the test object). Typical tasks are as follows:
  - Reviewing test plans and test cases
  - Using test tools and test monitoring tools (for example, to measure performance)
  - Executing and logging tests, including evaluating and documenting the results and detected deficiencies

In this context, what does the Certified Tester training offer? The basic *Certified Tester* training (Foundation Level) qualifies for the “tester” role (without covering the required IT basics). This means that a Certified Tester knows why discipline and structured work are necessary. Under the supervision of a test manager, a Certified Tester can manually execute and document tests. He or she is familiar with basic techniques for test specification and test management. Every software developer should also know these foundations of software testing to be able to adequately execute the testing tasks required by organizational models 1 and 2. Before someone is able to fulfill the role of a test designer or test manager, appropriate experience as a tester should be gathered. The second educational level (Advanced Level) offers training for the tasks of the designer and manager.

*Certified Tester*

---

1. The term tester is often also used as generic term for all the previously mentioned roles.

*Social competence is important*

To be successful, in addition to technical and test-specific skills, a tester needs social skills:

- Ability to work in a team, and political and diplomatic aptitude
- Skepticism (willingness to question apparent facts)
- Persistence and poise
- Accuracy and creativity
- Ability to get quickly acquainted with (complex fields of) application

*Multidisciplinary team*

Especially in system testing, it is often necessary to extend the test team by adding IT specialists, at least temporarily, to perform work for the test team. For example, these might be database administrators, database designers, or network specialists. Professional specialists from the application field of the software system currently being tested or the business are often indispensable. Managing such a multidisciplinary test team can be difficult even for experienced test managers.

*Specialized software test service providers*

If appropriate resources are not available within the company, test activities can be given to external software testing service providers. This is similar to letting an external software house develop software. Based on their experience and their use of predefined solutions and procedures, these test specialists are able to provide an optimal test for the project. They can also provide missing specialist skills from each of the previously mentioned qualification profiles for the project.

## 6.2 Planning

Testing should not be the only measure for quality assurance (QA). It should be used in combination with other quality assurance measures. Therefore, an overall plan for quality assurance is needed that should be documented in the quality assurance plan.

### 6.2.1 Quality Assurance Plan

Guidelines for structuring the quality assurance plan can be found in IEEE standard 730-2002 [IEEE 730-2002]. The following subjects shall be considered (additional sections may be added as required. Some of the material may also appear in other documents).

**Contents of a Software Quality Assurance Plan as defined in IEEE 730-2002:<sup>a</sup>**

1. Purpose
2. Reference documents
3. Management
4. Documentation
5. Standards, practices, conventions, and metrics
6. Software reviews
7. Test
8. Problem reporting and corrective action
9. Tools, techniques, and methodologies
10. Media control
11. Supplier control
12. Records collection, maintenance, and retention
13. Training
14. Risk management
15. Glossary
16. SQA Plan Change Procedure and History

- a. IEEE Standard 730 in its new form from 2013 [IEEE 730-2013] has a new title, Standard for Software Quality Assurance Processes, and does not contain a standard layout for a software quality assurance plan anymore.

During quality assurance planning, the role the tests play as special, analytical measures of quality control is roughly defined. The details are then determined during test planning and documented in the test plan.

### 6.2.2 Test Plan

A task as extensive as testing requires careful planning. This planning and test preparation starts as early as possible in the software project. The test policy of the organization and the objectives, risks, and constraints of the project as well as the criticality of the product influence the test plan.

The test manager might participate in the following planning activities:

*Test planning activities*

- Defining the overall approach to and strategy for testing (see section 6.4)
- Deciding about the test environment and test automation
- Defining the test levels and their interaction, and integrating the testing activities with other project activities

- Deciding how to evaluate the test results
- Selecting metrics for monitoring and controlling test work, as well as defining test exit criteria
- Determining how much test documentation shall be prepared and determining templates
- Writing the test plan and deciding on what, who, when, and how much testing
- Estimating test effort and test costs; (re)estimating and (re)planning the testing tasks during later testing work

The results are documented in the test plan. IEEE Standard 829-1998 [IEEE 829] provides a template.

#### **Test Plan according to IEEE 829-1998**

1. Test plan identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item pass/fail criteria (test exit criteria)
8. Suspension criteria and resumption requirements
9. Test deliverables
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Schedule
15. Risk and contingencies
16. Approvals

This structure<sup>2</sup> works well in practice. The sections listed will be found in real test plans in many projects in the same, or slightly modified, form. The

---

2. A detailed description of the listed points in IEEE 829-1998 can be found in Appendix A. The new standard [IEEE 829-2008] shows an outline for a master test plan and a level test plan. IEEE Standard 1012 ([IEEE 1012]) gives another reference structure for a verification and validation plan. This standard can be used for planning the test strategy for more complex projects.

new edition of IEEE 829-2008 [IEEE 829-2008] differentiates between “Master Test Plan” and “Level Test Plan.” The overall test plan (“Master Test Plan”) is required for every project. The different level test plans are optional, depending on the criticality of the product developed. An existing test plan according to IEEE 829-1998 can be changed into the structure of the master test plan in IEEE 829-2008 using mapping or a cross-reference listing. The new standard also has a different approach: There is an explicit requirement for tailoring the test documentation depending on product risks and organizational needs. The standard encourages putting some information from the plans into tools or, if necessary, other plans.

When preparing for an exam using the Foundation syllabus version 2015, IEEE Standard 829-2008, not 1998, should be studied!

Test planning is a continuous activity for the test manager throughout all phases of the development project. The test plan and related plans must be updated regularly, based on feedback from test activities and reacting to changing project risks.

### 6.2.3 Prioritizing Tests

Even with good planning and control, it is possible that the time and budget for the total test, or for a certain test level, are not sufficient for executing all planned test cases. In this case, it is necessary to select test cases in a suitable way. Even with a reduced number of executable test cases, it must be assured that as many as possible critical faults are found. This means test cases must be prioritized.

Test cases should be prioritized so that if any test ends prematurely, the best possible test result at that point of time is achieved.

Prioritization also ensures that the most important test cases are executed first. This way important problems can be found early.

The criteria for prioritization, and thus for determining the order of execution of the test cases, are outlined next. Which criteria are used depends on the project, the application area, and the customer requirements.

The following criteria for prioritization of test cases may be used:

- The **usage frequency** of a function or the **probability** of failure in software use. If certain functions of the system are used often and they contain a fault, then the probability of this fault leading to a failure is high. Thus, test cases for this function should have a higher priority than test cases for a less-often-used function.

*Prioritization rule*

*The most important test cases first*

*Criteria for prioritization*

- **Failure risk.** Risk is the combination (mathematical product) of severity and failure probability. The severity is the expected damage. Such risks may be, for example, that the business of the customer using the software is impaired, thus leading to financial losses for the customer. Tests that may find failures with a high risk get higher priority than tests that may find failures with low risks (see also section 6.4.3).
- The **visibility** of a failure for the end user is a further criterion for prioritization of test cases. This is especially important in interactive systems. For example, a user of a city information service will feel unsafe if there are problems in the user interface and will lose confidence in the remaining information output.
- Test cases can be chosen depending on the **priority of the requirements**. The different functions delivered by a system have different importance for the customer. The customer may be able to accept the loss of some of the functionality if it behaves wrongly. For other parts, this may not be possible.
- Besides the functional requirements, the **quality characteristics** may have differing importance for the customer. Correct implementation of the important quality characteristics must be tested. Test cases for verifying conformance to required quality characteristics get a high priority.
- Prioritization can also be done from the perspective of development or system architecture. Components that lead to severe consequences when they fail (for example, a crash of the system) should be tested especially intensively.
- **Complexity** of the individual components and system parts can be used to prioritize test cases. Complex program parts should be tested more intensively because developers probably introduced more faults. However, it may happen that program parts seen as easy contain many faults because development was not done with the necessary care. Therefore, prioritization in this area should be based on experience data from earlier projects run within the organization.
- Failures having a high **project risk** should be found early. These are failures that require considerable correction work that in turn requires special resources and leads to considerable delays of the project (see section 6.4.3).

In the test plan, the test manager defines adequate priority criteria and priority classes for the project. Every test case in the test plan should get a

priority class using these criteria. This helps in deciding which test cases can be left out if resource problems occur.

Where many faults were found before, more are present. This phenomenon occurs often in projects. To react appropriately, it must be possible to change test case priority. In the next test cycle (see section 6.5), additional test cases should be executed for such defect-prone test objects.

Without prioritizing test cases, it is not possible to adequately allocate limited test resources. Concentration of resources on high-priority test cases is a MUST.

*Where there are many defects, there are probably more*

### 6.2.4 Test Entry and Exit Criteria

Defining clear test entry and exit criteria is an important part of test planning. They define when testing can be started and stopped (totally or within a test level).

Here are typical criteria, or checkpoints, that need to be fulfilled before executing the planned tests:

- The test environment is ready.
- The test tools are ready for use in the test environment.
- Test objects are installed in the test environment.
- The necessary test data is available.

*Test start criteria*

These criteria are preconditions for starting test execution. They prevent the test team from wasting time trying to run tests that are not ready.

Exit criteria are used to make sure test work is not stopped by chance or prematurely. They prevent tests from ending too early, for example, because of time pressure or because of resource shortages. But they also prevent testing from being too extensive. Here are some typical exit criteria and corresponding metrics or indicators:

*Exit criteria*

- Achieved test coverage: Tests run, covered requirements, code coverage, etc.
- Product quality: Defect density, defect severity, failure rate, and reliability of the test object
- Residual risk: Tests not executed, defects not repaired, incomplete coverage of requirements or code, etc.
- Economic constraints: Allowed cost, project risks, release deadlines, and market chances

The test manager defines the project-specific test exit criteria in the test plan. During test execution, these criteria are then regularly measured and

evaluated and serve as the basis for decisions by test and project management.

## 6.3 Cost and Economy Aspects

Testing can be very costly and can constitute a significant cost factor in software development. How much effort is adequate for testing a specific software product? When is the test cost higher than the possible benefit?

To answer these questions, one must understand the potential defect costs due to lack of checking and testing. Then, one has to compare defect costs and testing costs.

### 6.3.1 Costs of Defects

If testing activities are reduced or cut out completely, there will be more undetected faults and deficiencies in the product. These remain in the product and may lead to the following costs:

*Costs due to product deficiencies*

- **Direct defect costs:** Costs that arise for the customer due to failures during operation of the software product (and that the vendor may have to pay for). Examples are costs due to calculation mistakes (data loss, wrong orders, damage of hardware or parts of the technical installation, damage to personnel); costs because of the failure of software-controlled machines, installations, or business processes; and costs due to installation of new versions, which might also require training employees. Very few people think of these costs, but they can be huge. The impact from just the time it takes to install a new version at all customer sites can be enormous.
- **Indirect defect costs:** Costs or loss of sales for the vendor that occur because the customer is dissatisfied with the product. Some examples include penalties or reduction of payment for failure to meet contractual requirements, increased costs for the customer hotline and support, bad publicity, even legal costs such as loss of license (for example, for safety critical software).
- **Costs for defect correction:** Costs paid to vendors for fault correction. For example, time needed for failure analysis, correction, retest and regression test, redistribution and reinstallation, new customer and user training, delay of new products due to tying up the developers with maintenance of the existing product, decreasing competitiveness.

It is hard to determine which types of costs will occur in reality, how likely it is, and how expensive it will be, that is, how high the failure cost risk is for a project. This risk depends of course on the kind and size of the software product, the type and business area of the customer, the design of the contract, legal constraints, the type of failures, and the number of installations or end users. There are certainly big differences between software developed specifically for a customer and commercial off-the-shelf products. In case of doubt, all these influencing factors must be evaluated in a project-specific risk analysis.

*Risk analysis*

It is crucial to find faults as early as possible after their creation. Defect costs grow rapidly the longer a fault remains in the product (one of the fundamental principles in chapter 2). This is independent of how high the risk of a fault really is.

*Finding defects as early as possible lowers the costs*

- A fault that is created very early (e.g., an error in the requirements definition) can, if not detected, produce many subsequent defects during the following development phases (“multiplication” of the original defect).
- The later a fault is detected, the more corrections are necessary. Previous phases of the development (requirements definition, design, and programming) may even have to be partly repeated.

A reasonable assumption is that with every test level, the correction costs for a fault double with respect to the previous level. More information on this can be found in [URL: NIST Report].

If the customer has already installed the software product, there is the additional risk of direct and indirect defect costs. In the case of safety-critical software (control of technical installations, vehicles, aircraft, medical devices, etc.), the potential consequences and costs can be disastrous.

### 6.3.2 Cost of Testing

The most important action to reduce or limit risk is to plan verification and test activities. But there are plenty of factors that influence the cost<sup>3</sup> of such testing activities, and in practice they are difficult to quantify. The following list shows the most important factors that a test manager should take into account when estimating the cost of testing:

---

3. A detailed discussion can also be found in [Pol 98] and [Pol 02].

- **Maturity<sup>4</sup>** of the development process
  - Stability of the organization
  - Developer's error rate
  - Change rate for the software
  - Time pressure because of unrealistic plans
  - Validity, stability, and correctness of plans
  - Maturity of the test process, and the discipline in configuration, change, and incident management
- **Quality and testability of the software**
  - Number, severity, and distribution of defects in the software
  - Quality, expressiveness, and relevance of the documentation and other information used as test basis
  - Size and type of the software and its system environment
  - Complexity of the problem domain and the software (e.g., cyclomatic number, see section 4.2.5)
- **Test infrastructure**
  - Availability of testing tools
  - Availability of test environment and infrastructure
  - Availability of and experience with testing processes, standards, and procedures
- **Employee (project member) qualification**
  - Tester experience and know-how about the field of testing
  - Tester experience and know-how about test tools and test environment
  - Tester experience and know-how about the test object
  - Collaboration between the tester, the developer, management, and customer
- **Quality requirements**
  - Intended test coverage
  - Intended reliability or maximum number of remaining defects after testing
  - Requirements for security and safety
  - Requirements for test documentation<sup>5</sup>

---

4. There are different methods to assess the maturity of software development processes. More information can be found in the ISTQB Advanced Test Manager syllabus [URL: ISTQB].

---

### ■ Test approach

- The testing objectives (themselves driven by quality requirements) and means to achieve them, such as number and comprehensiveness of test levels (component, integration, system test)
- The chosen test techniques (black box or white box)
- Test schedule (start and execution of the test work in the project or in the software life cycle)

The test manager can directly influence only a few of these factors. The manager's perspective looks like this:

*The test manager's influence*

### ■ Maturity of the software development process

This cannot be influenced in the short run; it is a given and must be accepted as is. Influence in this area can only be exercised in the long run, using a process improvement program.

### ■ Testability of the software

This is very dependent on the maturity of the development process. A well-structured process with reviews leads to better-structured software that is easier to test. This factor can only be influenced in the long run through a process improvement program.

### ■ Test infrastructure

Usually this is a given, but it may be improved during the project in order to save time and cost when it is used.

### ■ Qualification of the project members

This can be changed relatively fast by choosing different test personnel, but training may help in the longer run.

### ■ Quality goals

They are given by customers and other stakeholders and can be changed only slightly (by prioritization).

### ■ Test approach

This can be freely chosen and is the only way a test manager can control and monitor in the short run.

- 
5. Medical devices and other safety-critical applications require certification by regulation authorities like, for example, the FDA [URL: FDA]. Such certification follows standards, which require a certain level of test documentation.

### 6.3.3 Test Effort Estimation

Before defining a schedule and assigning resources, the test manager must estimate the testing effort to be expected.

*General estimation approaches*

For small projects, this estimation can be done in one step. For larger projects, separate estimations for each test level and test cycle may be necessary.

In general, two approaches for estimation of test effort are possible:

- Listing all testing tasks; then letting either the task owner or experts who have estimation experience estimate each task
- Estimating the testing effort based on effort data of former or similar projects, or based on typical values (e.g., average number of test cases run per hour)

The effort for every testing task depends on the factors described in the earlier section on testing costs (section 6.3.2). Most of these factors influence each other, and it is nearly impossible to analyze them completely. Even if no testing task is forgotten, task-driven test effort estimation tends to underestimate the testing effort. Estimating based on experience data of similar projects or typical values usually leads to better results.

*Rule of thumb*

If no data is available, the following rule of thumb can be helpful: testing tasks (including all test levels) in typical business application development costs about 50% of the overall project resources.

## 6.4 Choosing the Test Strategy and Test Approach

A test strategy or approach defines the project's testing objectives and the means to achieve them. It therefore determines testing effort and costs. Selecting an appropriate test strategy is one of the most important planning task decisions for a test manager. The goal is to choose a test approach that optimizes the relation between costs of testing and costs of possible defects as well as minimizes the risk (see section 6.4.3).

*Cost-benefit relationship*

The test costs should, of course, be less than the costs that would be caused by surviving defects and deficiencies in the final product. But, very few software development organizations possess or bother to collect data that enables them to quantify the relation between costs and benefits. This often leads to intuitive rather than rational decisions about how much testing is enough.

### 6.4.1 Preventative vs. Reactive Approach

The point in time at which testers become involved highly influences the approach. We can distinguish two typical situations:

- **Preventive approaches** are those in which testers are involved from the beginning: test planning and design start as early as possible. The test manager can really optimize testing and reduce testing costs. Use of the general V-model (see figure 3-1), including design reviews, etc., will contribute a lot to prevent defects. Early test specification and preparation, as well as application of reviews and static analysis, contribute to finding defects early and thus lead to reduced defect density during test execution. When safety-critical software is developed, a preventive approach may be mandatory.
- **Reactive approaches** are those in which testers are involved (too) late and a preventive approach cannot be chosen: test planning and design starts after the software or system has already been produced. Nevertheless, the test manager must find an appropriate solution even in this case. One very successful strategy in such a situation is called *exploratory testing*. This is a heuristic approach in which the tester “explores” the test object and test design, test execution, and evaluation occur nearly concurrently (see also section 5.3).

Preventative approaches should be chosen whenever possible. Cost analysis clearly shows the following:

- The testing process should start as early as possible in the project.
- Testing should continuously accompany all phases of the project.

*When should testing be started?*

---

In the VSR project, test planning and test specification started immediately after the requirements document was approved. For each requirement, at least one test case was designed. The draft test specification created using this approach was subjected to a review. Representatives for the customer, the development staff, and the later system test staff were involved in this review. The result was that many requirements were identified as “unclear” or “incomplete.” Additionally, staff found incorrect or insufficient test cases.

Therefore, simply preparing reasonable tests and discussing them with the developers and stakeholders helped to find many problems long before the first test was run.

---

**Example:**  
**VSR test planning**

### 6.4.2 Analytical vs. Heuristic Approach

During test planning and test design, the test manager may use different sources of information. Two extreme approaches are possible:

- **Analytical approach**

Test planning is founded on data and (mathematical) analysis of it. The criteria discussed in section 6.3 will be quantified (at least partially) and their correlation will be modeled. The amount and intensity of testing are then chosen such that individual or multiple parameters (costs, time, coverage, etc.) are optimized.

- **Heuristic approach**

Test planning is founded on experience of experts (from inside or outside the project) and/or on rules of thumb. Reasons may be that no data is available, mathematical modeling is too complicated, or the necessary know-how is missing.

The approaches used in practice are between these extremes and use (to different degrees) both analytical and heuristic elements:

- **Model-based testing** uses abstract functional models of the software under test for test case design, to find test exit criteria, and to measure test coverage. An example is state-based testing (see section 5.1.3), where state transition machines are used as models.
- **Statistical or stochastic (model-based) testing** uses statistical models about fault distribution in the test object, failure rates during use of the software (such as reliability growth models), or the statistical distribution of use cases (such as operational profiles) to develop a test approach. Based on this data, the test effort is allocated and test techniques are chosen.
- **Risk-based testing** uses information on project and product risks and directs testing to areas with high risk. This is described in more detail in the next section.
- **Process- or standard-compliant approaches** use rules, recommendations,<sup>6</sup> and standards (e.g., the V-model or IEEE 829) as a “cookbook.”
- **Reuse-oriented approaches** reuse existing test environments and test material. The goal is to set up testing quickly by maximal reuse.

---

6. Such recommendations contain a lot of heuristics and experience-based knowledge.

- **Checklist-based (methodical) approaches** use failure and defect lists from earlier test cycles,<sup>7</sup> lists of potential defects or risks,<sup>8</sup> or prioritized quality criteria and other less formal methods.
- **Expert-oriented approaches** use the expertise and “gut feeling” of involved experts (for the technology used or the application domain). Their personal feeling about the technologies used and/or usage domain influences and controls their choice of test approach.

These approaches are seldom used as they are described. Generally, a combination of several approaches is used to develop the testing strategy.

### 6.4.3 Testing and Risk

When looking for criteria to select and prioritize testing goals, test methods, and test cases, one of the best criteria is risk.

*Risk = damage × probability*

*Risk* is defined as the mathematical product of the loss or damage due to failure and the probability (or frequency) of failure resulting in such damage. Damage comprises any consequences or loss due to failure (see section 6.3.1). The probability of occurrence of a product failure depends on the way the software product is used. The software’s operational profile must be considered here. Therefore, detailed estimation of risks is difficult.<sup>9</sup> Risk factors to be considered may arise from the project (project risks) as well as from the product to be delivered (product risks).

Project risks are risks that threaten the project’s capability to deliver the product:

*Project risks*

- Supplier-side risks such as, for example, the risk that a subcontractor fails to deliver or fighting about the contract. Project delays or even legal action may result from these risks.
- An often-underestimated organizational risk is lack of necessary resources (total or partial lack of personnel with the necessary skills; recognizing necessary training but not implementing it), problems of human interaction (e.g., if testers or test results do not get adequate attention), or internal power struggles such as no or insufficient cooperation between different departments.

---

7. Where defects have been found before, more defects can usually be found! Defects are symptoms of further problems. For defect-prone areas, it is sensible to add extra tests in the following test cycles.

8. An analytical standard method for this is failure mode and effects analysis (FMEA).

9. A spreadsheet-based method for estimating risks or risk classes can be found at [URL: Schaefer].

- Technical problems are another project risk. Wrong, incomplete, fuzzy, or infeasible requirements may lead to project failure. If new technologies, tools, programming languages, or methods are applied without sufficient experience, the expected result—to get better results faster—may easily turn into the opposite. Another technical project risk is that the quality of intermediate results is too low (design documents, program code, or test cases) or that defects have not been detected and corrected. There are even risks for the test itself—for example, if the test environment is not ready or the test data is incomplete.

*Product risks* Product risks are risks resulting from problems with the delivered product:

- The delivered product has inadequate functional or nonfunctional quality. Or the quality of the data to be processed is poor (for example, because of faults in previous data migration or conversion).
- The product is not fit for its intended use and is thus unusable.
- The use of the product causes harm to equipment or even endangers human life.

*Risk management* The [IEEE 730] and [IEEE 829] standards for quality assurance and test plans demand systematic risk management. This comprises the following actions:

- Regularly identifying what can go wrong (risks)
- Prioritizing identified risks
- Implementing actions to mitigate or fight those risks

An important risk mitigation activity is testing; testing provides information about existing problems and the success or failure of correction. Testing decreases uncertainty about risks, helps to estimate risks, and identifies new risks.

*Risk-based Testing* Risk-based testing helps to minimize and fight product risks from the start of the project. Risk-based testing uses information about identified risks for planning, specification, preparation, and execution of the tests. All major elements of the test approach are determined based on risk:

- The test techniques to be used
- The extent of testing
- The priority of test cases

Even other risk-minimizing measures, such as training for inexperienced software developers, are considered as supplements to measures for testing.

Risk based prioritization of the tests ensures that risky product parts are tested more intensively and earlier than parts with lower risk. Severe problems (causing much corrective work or serious delays) are found as early as possible. Opposed to this, distributing scarce test resources equally throughout all test objects does not make much sense because this approach will test critical and uncritical product parts with the same intensity. Critical parts are then not adequately tested and test resources are wasted on uncritical parts.

*Risk-based test prioritization*

## 6.5 Managing The Test Work

Every cycle through the testing process (see section 2.2, figure 2-4) usually results in tasks for correction or →changes for the developers. When bugs are corrected or changes are implemented, a new version of the software comes into life, and it must be tested. In every test level, the test process is repeatedly executed.

The test manager has to initiate these test cycles, monitor their progress, and control the test work. Depending on the size of the project, a test level may be managed by its own test manager.

*Test manager tasks*

### 6.5.1 Test Cycle Planning

Section 6.2 described the initial test planning (test approach and general work flow). This should be developed early in a project and described in the test plan.

This general plan must be detailed in a detailed plan for the concrete test cycle to be run next, and it must be adapted to the current project situation. The following points should be addressed:

*Detailed planning  
per test cycle*

- **State of development:** The software available at the start of the test cycle may have less or different functionality than originally planned for. The test specification and test cases may need to be adapted.
- **Test results:** Problems discovered in earlier test cycles might require changed test priorities. Fixed defects require additional confirmation tests (retests), and these must be planned. Additional tests may be necessary because some problems may be difficult or impossible to reproduce or analyze.
- **Resources:** The plan for the current test cycle must be consistent with the project plan. Attention must be given to personal disposition planning, holiday planning, availability of test environment, and special tools.

*Planning the test effort* Using these preconditions, the test manager estimates effort and duration of the test work and plans in detail which test cases shall be executed by which tester, in which order, and at which points of time. The result of this detailed planning is the plan for the next test cycle or regression test cycle.

### 6.5.2 Test Cycle Monitoring

To measure and monitor the results of the ongoing tests, objective →test metrics should be used. They are defined in the test plan. Only metrics that are reliably, regularly, and simply measurable<sup>10</sup> should be used. These approaches are possible:

*Metrics for monitoring the test process*

- **Fault- and failure-based metrics**

Number of encountered faults and number of generated incident reports (per test object) in the particular release. This should also include the problem class and status, and, if possible, a relation to the size of the test object (lines of code), test duration, or other measures (see section 6.6).

- **Test-case-based metrics**

Number of test cases in a certain state, like specified or planned, →blocked (e.g., because of a fault not being eliminated), number of test cases run (passed or failed).

- **Test-object-based metrics**

Coverage of code, dialogs, possible installation variants, platforms, etc.

- **Cost-based metrics**

Test cost until now, cost of the next test cycle in relation to expected benefit (prevented failure cost or reduced project or product risk).

*Test status report*

The test manager lists the current measurement results in the test reports. After each test cycle, a test status report should show the following information about the status of the test activities:

- Test object(s), test level, test cycle date from ... to ...
- Test progress: tests planned/run/blocked
- Incident status: new/open/corrected
- Risks: new/changed/known
- Outlook: planning of the next test cycle

---

10. This is the case when the applied test tools automatically provide such data.

- **Assessment:** (subjective) assessment of the maturity of the test object, the possibility for release, or the current confidence

A template for such a report can be found in [IEEE 829].

On the one hand, the measured data serves as a means to determine the current situation and to answer the question, How far has the test progressed? On the other hand, the data serves as exit criterion and to answer the question, Can the test be finished and the product be released? The quality requirements to be met (the product's criticality) and the available test resources (time, personnel, test tools) determine which criteria are appropriate for determining the end of the test. The test exit criteria for the current project are also documented in the test plan. It should be possible to decide about each test exit criterion based on the collected test metrics.

*Test exit criteria*

The test cases in the VSR project are divided into the following three priority levels:

Priority	Meaning
1	Test case must be executed
2	Test case should be executed
3	Test case may be executed

**Example:**  
**Test completion criteria for the VSR-System test**

Based on this prioritization, the test plan describes the following decision about the test-case-based completion criteria for the VSR-System test:

- All test cases with priority 1 have been executed without failure.
- At least 60% of the test cases with priority 2 have been executed.

If the defined test exit criteria are met, project management (using advice from the test manager) decides whether the corresponding test object should be released and delivered. For component and integration testing, "delivery" means passing the test object to the next test level. The system test precedes the release of the software for delivery to the customer. Finally, the customer's acceptance test releases the system for operation in the real application environment.

*Product release*

Release does not mean "bug free." The product will surely contain some undiscovered faults, as well as some known ones that were rated as "not preventing release" and that therefore were not corrected. The latter faults are recorded in the incident database (also called →defect database or →problem database) and may be corrected later, during software maintenance (see section 3.6.1).

### 6.5.3 Test Cycle Control

*React on deviations  
from the plan*

If testing is delayed with respect to the project and test planning, the test manager must take suitable countermeasures. This is called test (cycle) control. These actions may relate to the test or any other development activity.

It may be necessary to request and deploy additional test resources (personnel, workstations, and tools) in order to compensate for the delay and catch up on the schedule in the remaining cycles.

If additional resources are not available, the test plan must be adapted. Test cases with low priority will be omitted. If test cases are planned in several variants, a further option is to only run them in a single variant (for example, tests are performed on one operating system instead of several). Although these adjustments lead to omission of some interesting tests, the available resources can at least make it possible to execute the high-priority test cases.

Depending on the severity of the faults and problems found, test duration may be extended. This happens because additional test cycles become necessary, because the corrected software must be retested after each correction cycle (see section 3.7.4). This could mean that the product release must be postponed.

*Changes to test plan must be  
communicated clearly*

It is important that the test manager documents and communicates every change in the plan because the change in the test plan may increase the release risk (product risk). The test manager is responsible for communicating this risk openly and clearly to the people responsible for the project.

## 6.6 Incident Management<sup>11</sup>

To ensure reliable and fast elimination of failures detected by the various test levels, a well-functioning procedure for communicating and managing those incident reports is needed. Incident management starts during test execution or upon test cycle completion by evaluating the test log.

---

11. *Incident management* is called →*defect management* in the ISTQB advanced test manager syllabus

### 6.6.1 Test Log

After each test run, or at the latest upon completion of a test cycle, the test logs are evaluated. Real results are compared to the expected results. If the test was automated, the tool will normally do this comparison immediately. Each significant, unexpected event that occurred during testing could be an indication of a test object malfunctioning. Corresponding passages in the test log are analyzed. The testers ascertain whether a deviation from the predicted outcome really has occurred or whether an incorrectly designed test case, incorrect test automation, or incorrect test execution caused the deviation (testers, too, can make mistakes).

*Test log analysis*

If the test object caused the problem,<sup>12</sup> a defect or incident report is created. This is done for every unexpected behavior or observed deviation from the expected results found in the test log. An observation may be a duplicate of an observation recorded earlier. In this case, it should be checked to see whether the second observation yields additional information, which may make it possible to more easily search for the cause of the problem. Otherwise, to prevent duplication of an incident record, the same incident should not be recorded a second time.

*Documenting incidents*

However, the testers do not have to investigate the cause of a recorded incident. This (*debugging*) is the developers' responsibility.

*Cause analysis is a developer task*

### 6.6.2 Incident Reporting

In general, a central database is established for each project, in which all incidents and failures discovered during testing (and possibly during operation) are registered and managed. All personnel involved in development as well as customers and users can report incidents.<sup>13</sup> These reports can refer to problems in the tested (parts of) programs as well as to faults in specifications, user manuals, or other documents.

Incident reporting is also referred to as problem, anomaly, defect, or failure reporting. Not every incident or problem is due to a developer mistake. *Incident reporting* sounds less like an "accusation." Incident reporting is not a one-way street because every developer can comment on reports—for example, by requesting comments or clarification from a tester or by

---

12. Creating an incident report may of course also be useful if the tester caused the problem; for example, if the problem calls for further analyses. In this case, the incident will be directed to the tester and not to the developers.

13. This discussion focuses on communication between testers and developers instead of users.

rejecting an unjustified report. Should a developer correct a test object, the corrections will also be documented in the incident database. This enables the responsible tester to understand this correction's implications in order to retest it in the following test cycle.

At any point in time, the incident database enables the test manager and the project manager to get an up-to-date and complete picture of the number and state of problems and about the progress of corrections. For this purpose, the database should offer appropriate possibilities for reporting and analysis.

**Hint:**

**Use an incident database**

- One of the first steps when introducing a systematic test process for a project should be implementing disciplined incident management. An efficient incident database, giving role-related access to all staff involved in the project, is essential.

*Standardized reporting format*

To allow for smooth communication and to enable statistical analysis of the incident reports, every report must follow a project-wide unique report template. The test manager should define this template and reporting structure in, for example, the test plan.

In addition to the description of the problem, the incident report typically contains information identifying the tested software, test environment, name of the tester, and defect class and prioritization as well as other information that's important for reproducing and localizing the fault. Table 6-1 shows an example of an incident report template.

A similar, slightly less complex structure can be found in [IEEE 829]. Or a report can include many additional attributes and more detail, as shown in [IEEE 1044].

If the incident database is used in acceptance testing or product support, additional customer data must be collected. The test manager has to develop a template or scheme suitable for the particular project.

*Document all information relevant to reproduction and correction*

In doing so, it is important to collect all information necessary for reproducing and localizing a potential fault as well as information enabling analysis of product quality and correction progress.

Irrespective of the scheme agreed upon, the following rule must be observed: Each report must be written in such a way that the responsible developer can identify the problem with minimal effort and find its cause as fast as possible. Reproducing problems, localizing the cause of problems, and repairing faults are usually unplanned extra work for developers. Thus, the tester has the task of "selling" the incident report to the

developers. In this situation, it is very tempting for developers to ignore or postpone analysis and repair of problems, which are unclearly described or difficult to understand.

	Attribute	Meaning
Identification	Id / Number	Unique identifier/number for each report
	Test object	Identifier or name of the test object
	Version	Identification of the exact version of the test object
	Platform	Identification of the HW/SW platform or the test environment where the problem occurs
	Reporting person	Identification of the reporting tester (possibly with test level)
	Responsible developer	Name of the developer or the team responsible for the test object
	Reporting date	Date and possibly time when the problem was observed
Classification	Status	The current state (and complete history) of processing for the report (section 6.6.4)
	Severity	Classification of the severity of the problem (section 6.6.3)
	Priority	Classification of the priority of correction (section 6.6.3)
	Requirement	Pointer to the (customer-) requirements which are not fulfilled due to the problem
	Problem source	The project phase, where the defect was introduced (analysis, design, programming); useful for planning process improvement measures
Problem description	Test case	Description of the test case (name, number) or the steps necessary to reproduce the problem
	Problem description	Description of the problem or failure that occurred; expected vs. actual observed results or behavior
	Comments	List of comments on the report from developers and other staff involved
	Defect correction	Description of the changes made to correct the defect
	References	Reference to other related reports

**Table 6-1***Incident report template*

### 6.6.3 Defect Classification

An important criterion for managing a reported problem is its severity, that is, how far product use is impaired. The degree of severity will certainly be different for 100 open defect reports concerning system crashes

in the database than it would be with layout errors in windows. Severity can be classified using the classes given in table 6-2.

**Table 6-2**  
Failure severity

Class	Description
1 – FATAL	System crash, possibly with loss of data. The test object cannot be released in this form.
2 – VERY SERIOUS	Essential malfunctioning; requirements not adhered to or incorrectly implemented; substantial impairment to many stakeholders. The test object can only be used with severe restrictions (difficult or expensive workaround).
3 – SERIOUS	Functional deviation or restriction (“normal” failure); requirement incorrectly or only partially implemented; substantial impairment to some stakeholders. The test object can be used with restrictions.
4 – MODERATE	Minor deviation; modest impairment to few stakeholders. System can be used without restrictions.
5 – MILD	Mild impairment to few stakeholders; system can be used without restrictions. For example, spelling errors or wrong screen layout.

The severity of a problem should be assigned from the point of view of the user or future user of the test object. The classifications in table 6-2, however, do not indicate how quickly a particular problem should be corrected. Priority associated with handling the problem (→failure priority) is a different matter and should not be confused with severity! When determining the priority of corrections, additional requirements defined by product or project management (for example, correction complexity), as well as requirements about further test execution (blocked tests), must be taken into account. Therefore, the question of how quickly a fault should be corrected is answered by an additional attribute, *fault priority* (or rather, *correction priority*). Table 6-3 presents a possible classification.

**Table 6-3**  
Fault priority

Priority	Description
1 – IMMEDIATE	The user’s business or working process is blocked or the running tests cannot be continued. The problem requires immediate, or if necessary, provisional repair (→“patch”).
2 – NEXT RELEASE	The correction will be implemented in the next regular product release or with the delivery of the next (internal) test object version.
3 – ON OCCASION	The correction will take place when the affected system parts are due for a revision anyway.
4 – OPEN	Correction planning has not taken place yet.

Analyzing the severity and priority of reported incidents allows the test manager to make statements about product robustness or deliverability. Apart from test status determination and clarification of questions relating to how many faults were found, how many of them are corrected, and how many are still to be corrected, trend analyses are important. This means making predictions based on the analysis of the trend of incoming incident reports over time. In this context, the most important question is whether the volume of product problems still increases or whether the situation seems to improve.

*Incident analysis for  
controlling the test process*

Data from incident reports can also be used to improve the test process; for example, a comparison of data from several test objects can demonstrate which test objects show an especially small number of faults. This could mean a lack of tests or that the program has been implemented especially carefully.

*Incident analysis for  
improving the test process*

#### **6.6.4 Incident Status**

Test management not only has a responsibility to make sure incidents are collected and documented properly but is additionally responsible (in cooperation with project management) for enabling and supporting rapid fault correction and delivery of improved versions of the test object.

This necessitates continuous monitoring of the defect analysis and correction process. For this purpose the incident status is used. Every incident report (see table 6-1) passes a series of predefined states, covering all steps from original reporting to successful defect resolution. Table 6-4 shows an example for an incident status scheme. Figure 6-2 demonstrates this procedure.

A crucial fact that is often ignored is that only the tester may set the state to “Closed” and not the developer! And this should happen only after the repeated test (retest) has proven that the problem described in the problem report does not occur anymore. Should new failures occur as side effects after bugs are fixed, these failures should be reported in new incident reports.

*Only the tester may set the  
state to “Closed”*

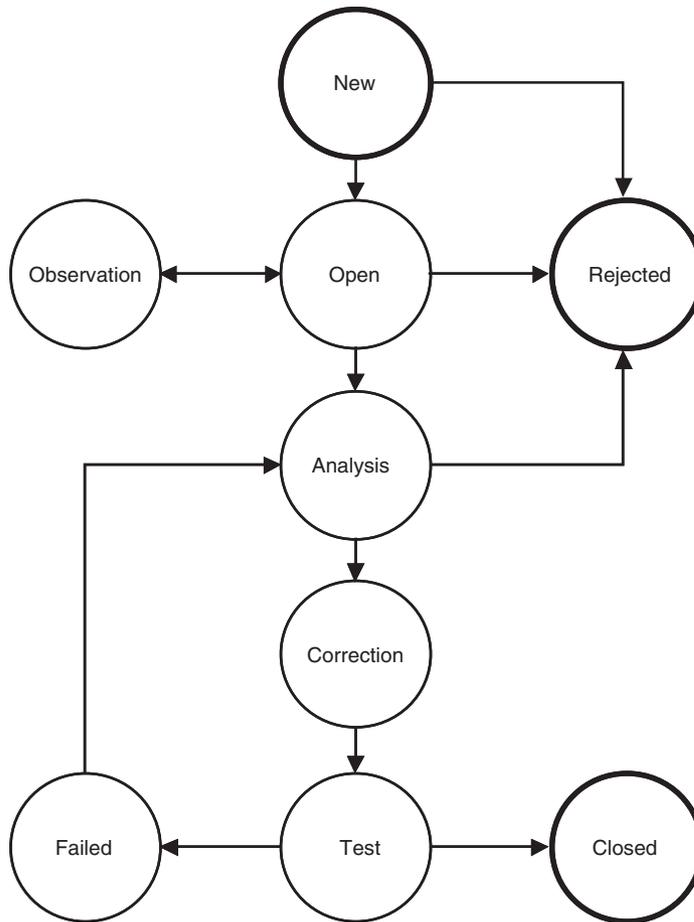
**Table 6-4**  
*Incident status scheme*

Status (set by)	Description
<b>New</b> (Tester)	A new report was written. The person reporting has included a sensible description and classification.
<b>Open</b> (Test manager)	The test manager regularly checks the new reports on comprehensibility and complete description of all necessary attributes. If necessary, attributes will be adjusted to ensure a project-wide uniform assessment. Duplicates or obviously useless reports are adjusted or rejected. The report is assigned to a responsible developer and its status is set to "Open."
<b>Rejected</b> (Test manager)	Duplicated or clearly wrong or unjustified incidents are rejected (no fault in the test object, request for change not taken into account).
<b>Analysis</b> (Developer)	As soon as the responsible developer starts processing this report, the status is set to "Analysis." The result of the analysis (cause, possible remedies, estimated correction effort, etc.) will be documented in comments.
<b>Observation</b> (Developer)	The incident described can neither be reconstructed nor be eliminated. The report remains outstanding until further information/insights are available.
<b>Correction</b> (Project manager)	Based on the analysis, the project manager decides if correction should take place and therefore sets the status to "Correction." The responsible developer performs the corrections and documents the kind of corrections done using comments.
<b>Test</b> (Developer)	As soon as the responsible developer has corrected the problem from his point of view, the report is set to "Test" status. The new software version containing this correction is identified.
<b>Closed</b> (Tester)	Reports carrying the status "Test" are verified in the next test cycle. For this purpose, at least the test cases, which discovered the problem, are repeated. Should the test confirm that the repair was successful, the tester finishes the report-history by setting the final status "Closed."
<b>Failed</b> (Tester)	Should the repeated test show that the attempt to repair was unsuccessful or insufficient, the status is set to "Failed" and a repeated analysis becomes necessary.

**Example of extended  
test exit criteria for the  
VSR-System test**

The test exit criteria for the VSR-System test shall reflect not only test progress but also the accomplished product quality. Therefore, the test manager enhances the test exit criteria with fault-based metrics as follows:

- All faults of severity "1 - FATAL" are "Closed."
- All faults of severity "2 - SEVERE" are "Closed."
- The number of "new" incident reports per test week is stable or falling.



**Figure 6-2**  
Incident status model

The scheme described previously can be applied to many projects. However, the model must be tailored to cover existing or necessary decision processes in the project. In the basic model, all decisions lie with one single person. In larger-scale projects, groups make the decisions. The decision processes grow more complex because representatives of many stakeholders must be heard.

In many cases, changes to be done by the developers are not really fault corrections, but real (functional) enhancements. Because the distinction between “incident report” and “enhancement request” and the rating as “justified” or “not justified” is often a matter of opinion, an institution accepting or rejecting incident reports and →change requests is needed.

*Change control board*

This institution, called the *change control board*, usually consists of representatives from the following stakeholders: product management, project management, test management, and the customer.

## 6.7 Requirements to Configuration Management

A software system consists of a multitude of individual components that must fit together to ensure the functionality of the system as a whole. In the course of the system's development, new, corrected, or improved versions or variants of each of these components evolve. Because several developers and testers take part in this process simultaneously, it is far from easy to keep track of the currently valid components and their relationships.

*Typical symptoms of  
insufficient configuration  
management*

If configuration management is not done properly in a project, the following typical symptoms may be observed:

- Developers mutually overwrite each other's modifications in the source code or other documents because simultaneous access to shared files is not avoided.
- Integration activities are impeded:
  - Because it is unclear which code versions of a specific component exist in the development team and which ones are the current versions
  - Because it is unclear which versions of several components belong together and can be integrated to a larger subsystem
  - Because different versions of compilers and other development tools are used
- Problem analysis, fault correction, and regression tests are complicated:
  - Because it is unknown where and why a component's code was changed with respect to a previous version
  - Because it is unknown from which code files a particular integrated subsystem (object code) originates
- Tests and →test evaluation are impeded:
  - Because it is unclear which test cases belong to which version of a test object
  - Because it is unclear which test cycle of which version of the test object gave which test results

Insufficient configuration management thus leads to a number of possible problems disturbing the development and test process. If, for example, it is unclear during a test level whether the examined test objects are the latest version, the tests rapidly lose any significance. A test process cannot be properly executed without reliable configuration management.

From the perspective of the test, the following requirements should be met:

#### ■ **Version management**

This is the cataloguing, filing, and retrieval of different versions of a →configuration item (for example, version 1.0 and 1.1 of a component consisting of several files). This also includes securing comments on the reason for the particular change.

#### ■ **Configuration identification**

This is the identification and management of all files (configuration objects) in the particular version, which together comprise a subsystem (configuration). The prerequisite for this is version management.

#### ■ **Incident and change status control**

This is the documenting of incident reports and change requests and the possibility to reconstruct their application on the configuration objects.

#### ■ **Configuration audits**

To check the effectiveness of configuration management, it is useful to organize configuration audits. Such an →audit offers the possibility to check whether the configuration management documented all software components, whether configurations can be correctly identified, etc.

---

The software developed in the VSR project is available in different languages (for example, English, German, and French) and must be compatible with several hardware and software platforms. Several components must be compatible with particular external software versions (e.g., the mainframe's current communication software). Furthermore, data from miscellaneous sources must be imported at regular intervals (e.g., product catalogues, price lists, and contract data) with changing content and format during the system's life cycle. The VSR configuration management must ensure that development and testing always have consistent, valid product configurations. Similar requirements exist during system operation at the customer.

---

*Testing depends on configuration management*

*Requirements to configuration management*

***Example of configuration management in the VSR project***

To implement configuration management conforming to the requirements mentioned earlier, differing processes and tools should be chosen depending on project characteristics. A configuration management plan must therefore determine a process tailored to the project situation. A standard for configuration management and respective plans can be found in [IEEE 828].

## 6.8 Relevant Standards

Today, a multitude of standards exist, setting constraints and defining the “state-of-the-art” even for software development. This is especially true for the area of software quality management and software testing, as the standards quoted in this book prove. One of the tasks for a quality manager or test manager is defining, in this context, which standards, rules, or possible legal directives are relevant for the product to be tested (product standards) or for the project (project standards) and to ensure that they are adhered to. Here are some possible sources of standards:

### ■ **Company standards**

These are company internal directives, procedure, and guidelines (for the supplier, but also possibly set by the customer), such as a quality management handbook, a test plan template, or programming guidelines.

### ■ **Best practices**

These are not standardized, but professionally accepted methods and procedures representing the state of the art in a particular field of application.

### ■ **Quality management standards**

These are standards spanning several industrial sectors, specifying minimal process requirements yet not stating specific requirements for process implementation. A well-known example is [ISO 9000], which requires appropriate (intermediate) tests during the production process (also in the special case of the software development process) without indicating when and how these tests are to be performed.

### ■ **Standards for particular industrial sectors**

These are standards defining for a particular product category or application field the minimum extent to which tests must be performed or

documented. An example is standard [RTCA-DO 178] for airborne software products; another example is [EN 50128] for railway signaling applications.

#### ■ **Software testing standards**

These are process or documentation standards, defining independently of the product how software tests should be performed; for example, the standards [BS 7925-2], [IEEE 829], [IEEE 1028], [ISO 29119].

The important and relevant standards for software testing are covered in this book. The test plan according to [IEEE 829-1998] and [IEEE 829-2008] is described in detail in appendix A. Following such standards makes sense, even when compliance is not mandatory. At least when encountering legal disputes, demonstrating that development has been done according to the “state of best industry practice” is helpful. This also includes compliance to standards.

## **6.9 Summary**

- Development activities and testing activities should be independently organized. The clearer this separation, the more effective the testing.
- Depending on the task to be executed within the test process, people with role-specific testing skills are needed. In addition to professional skills, social competence is required.
- The test manager’s tasks comprise the initial strategy and planning of the tests as well as further planning, monitoring, and controlling of the different test cycles.
- In the test plan, the test manager describes and explains the test strategy (test objectives, test approach, tools, etc.). The international standard [IEEE 829] provides a checklist for format and content.
- Faults and deficiencies that are not found by the testing and thus remain in the product can lead to very high costs. The test strategy has to balance testing costs, available resources, and possible defect costs.
- It is important to quickly decide which tests can be left out if lack of test resources occurs. To achieve this, the tests should be prioritized.
- Risk is one of the best criteria for prioritizing. Risk-based testing uses information about identified risks for planning and controlling all steps in the test process. All major elements of the test strategy are determined based on risk.

- Measurable test exit criteria objectively define when testing can be stopped. Without given test exit criteria, testing might stop randomly.
- Incident management and configuration management, together, form the basis for an efficient test process.
- Problem reports must be collected in a project-wide standardized way and followed up through all stages of the incident analysis and fault resolution process.
- Standards contain specifications and recommendations for professional software testing. Following such standards makes sense, even when compliance is not mandatory.

---

## 7 Test Tools

*This chapter gives an overview of the different test tools. Topics include how to choose and introduce these tools and the preconditions for using them.*

Due to the fast development of new testing tools, this chapter will undergo a major revision in the ISTQB syllabus version 2015. A reader preparing for an exam based on the new syllabus is advised to study the ISTQB syllabus in addition to this chapter.

### 7.1 Types of Test Tools

Test tools are normally used for these purposes:

*Why tools?*

- Improving test efficiency. Manual work, such as repetitive and time-consuming tasks, can be automated. Static analysis and test execution are examples of tasks that can be automated.
- Enabling tests. Tools may make it possible to execute tests that are impossible to do manually. This includes performance and load tests and tests of real-time inputs for control systems.
- Improving test reliability. Reliability is improved by automating manual tasks like comparing large amounts of data or simulating program behavior.

There are tools that accomplish one single task as well as tools that have several purposes. Tools for separate test execution, test execution automation, and generating or migrating test data belong to the first group. When tools have several capabilities, they are often called tool suites. Such a suite may, for example, automate test execution, logging, and evaluation.

*Test framework* *Test framework* is a term often used in discussions of test tools. In practice, it has at least three meanings:

- Reusable and extensible testing libraries that can be used to build testing tools (sometimes called test harnesses)
- The way of designing the test automation (e.g., data-driven, keyword-driven)
- The whole process of test execution

For the purpose of the ISTQB Foundation Level syllabus, the terms *test framework* and *test harness* are used interchangeably and are defined by the first two meanings.

*CAST tools* Test tools are often called CAST tools (Computer Aided Software Testing), which is derived from the term *CASE tools* (Computer Aided Software Engineering).

Depending on which activities or phases in the test process (see section 2.2) are supported, several tool types<sup>1</sup> or classes may be distinguished. In most cases, special tools are available within a tool class for special platforms or application areas (e.g., performance testing tools for the testing of web applications).

Only in very few cases is the whole range of testing tools used in a project. However, the available tool types should be known. The tester should be able to decide if and when a tool may be used productively in a project.

*Tool list on the Internet* A list of available test tools with suppliers can be found at [URL: Tool-List]. The following sections describe which functions the tools in the different classes provide.

### 7.1.1 Tools for Management and Control of Testing and Tests

*Test management* Test management tools provide mechanisms for easy documentation, prioritization, listing, and maintenance of test cases. They allow the tester to document and evaluate if, when, and how often a test case has been executed. They also facilitate the documentation of the results (“OK,” “not OK”). In addition, some tools support project management within testing (e.g., timing and use of resources). They help the test manager plan the tests and keep an overview of hundreds or thousands of test cases.

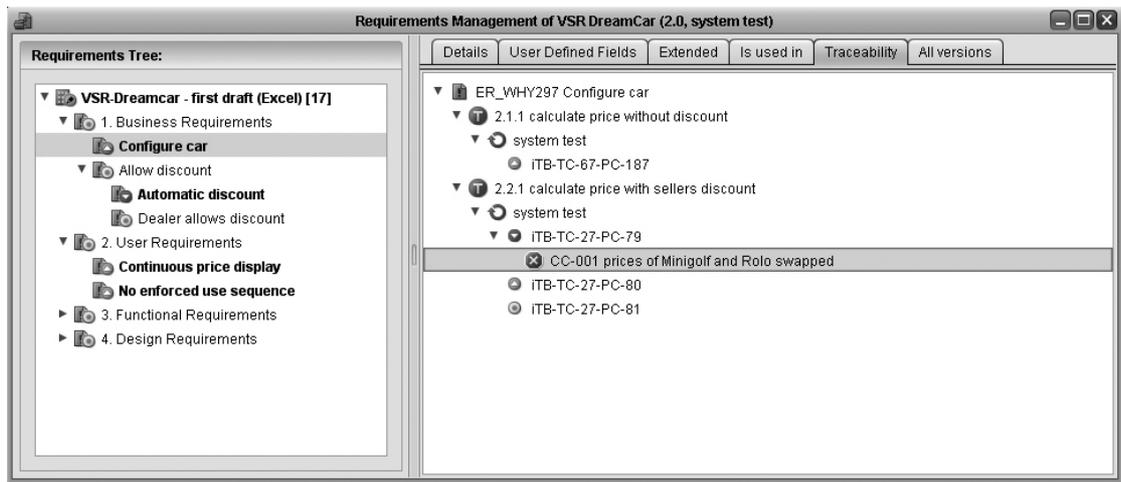
---

1. Commercial tools often support several activities or phases and may be assigned to several tool classes.

Advanced test management tools support requirements-based testing. In order to do this, they capture system requirements (or import them from requirements management tools) and link them to the tests, which test the corresponding requirements. Different consistency checks may be done; for example, a test may check to see if there is at least one planned test case for every requirement.

Figure 7-1 shows what this may look like for the CarConfigurator.

**Figure 7-1**  
Requirements-based test  
planning using TestBench  
[URL: TestBench]



The requirement *Configure car* has four assigned test cases: iTB-TC-67-PC-187, iTB-TC-27-PC-79, iTB-TC-27-PC-80 and iTB-TC-27-PC-81. iTB-TC-27-PC-79 found a failure and is connected to the corresponding incident report, CC-001. Test case iTB-TC-27-PC-79 is not yet approved. Thus, the tool marks the requirement *Configure car* as being only “partially tested,” using a corresponding icon in the requirements tree.

Tools for requirements management store and manage information about requirements. They allow testers to prioritize requirements and to follow their implementation status.

In a narrow sense, they are not testing tools, but they are very useful for defining a test based on requirements (see section 3.7.1) and for planning the test; for example, the test could be oriented on the implementation status of a requirement. For this purpose, requirements management tools are usually able to exchange information with test management tools. Thus, it is possible to seamlessly interconnect requirements, tests, and test

*Requirements management*

results. For every requirement the corresponding tests can be found and vice versa. The tools also help to find inconsistencies or holes in the requirements, and they can identify requirements without test cases; that is, requirements that otherwise might go untested.

*Incident management*

A tool for incident, problem, or failure management is nearly indispensable for a test manager. As described in section 6.6, incident management tools are used to document, manage, distribute, and statistically evaluate incident and failure messages. Better tools of this class support problem status models capable of being individually tailored. The whole work process may be defined, from the incident discovery through correction until the regression test has been executed. Every project member will be guided through the process corresponding to his role in the team.

*Configuration management*

Configuration management tools (see section 6.7) are, strictly speaking, not testing tools. They make it possible to keep track of different versions and builds of the software to be tested as well as documentation and testware. It is thus easier to trace which test results belong to a test run for a certain test object of a certain version.

*Tool integration*

Integrating test tools with test tools and with other tools is getting more and more important. The test management tool is the key for this:

- Requirements are imported from the requirements management tool and used for test planning. The test status for every requirement can be watched and traced in the requirements management tool or the test management tool.
- From the test management tool, test execution tools (for example, → test robots) are started and supplied with test scripts. The test results are automatically sent back and archived.
- The test management tool is coupled with the incident management tool. Thus, a plan for retest can be generated; that is, a list of all test cases necessary to verify which defects have been successfully corrected in the latest test object version.
- Finally, through configuration management, every code change is connected to the triggering incident or the change request causing it.

Such a tool chain makes it possible to completely trace the test status from the requirements through the test cases and test results to the incident reports and code changes.

*Generation of test reports  
and test documents*

Both test management and incident management tools include extensive analysis and reporting features, including the possibility to generate

the complete test documentation (test plan, test specification, test summary report) from the data maintained by the tool.

Usually, the format and contents of such documents can be individually adjusted. Thus the generated test documents will be easy to seamlessly integrate into the existing documentation workflow.

The collected data can be evaluated quantitatively in many ways. For example, it is very easy to determine how many test cases have been run and how many of them were successful or how often the tests have found failures or faults of a certain category. Such information helps to assess the progress of the testing and to manage the test process.

### 7.1.2 Tools for Test Specification

In order to make test cases reproducible, the pre- and postconditions as well as test input data and expected results need to be specified.

So-called test (data) generators can support the test designer in generating test data. According to [Fewster 99], several approaches can be distinguished, depending on the test basis used for deriving the test data:

*Test data generators*

- **Database-based test data generators** process database schemas and are able to produce test databases from these schemas. Alternatively, they perform dedicated filtering of database contents and thus produce test data. A similar process is the generation of test data from files in different data formats.
- **Code-based test data generators** produce test data by analyzing the test object's source code. A drawback and limitation is that no expected results can be generated (a test oracle is needed for this) and that only existing code can be covered (as with all white box methods). Faults caused by missing program instructions (missing code) remain undetected. The use of code as a test basis for testing the code itself is in general a very poor foundation. However, this approach may be used to generate a regression or platform test of an existing and reliable system, where the test result is collected for the existing system and platform and the generated test is then re-executed on other platforms or new versions.
- **Interface-based test data generators** analyze the test object's interface and identify the interface parameter domains and use. For example, equivalence class partitioning and boundary value analysis can be used to derive test data from these domains. Tools are available for analyzing different kinds of interfaces, ranging from application programming

interfaces (APIs) to graphical user interfaces (GUIs). The tool is able to identify which data fields are contained in a screen (e.g., numeric field, date) and generate test data covering the respective value ranges (e.g., by applying boundary value analysis). Here, too, the problem is that no expected results can be generated. However, the tools are very well suited for automatic generation of negative tests because specific target values are of no importance here; what is important is whether the test object produces an error message or not.

- **Specification-based test data generators** use a specification to derive test data and corresponding expected results. A precondition is that the specification is available in a formal notation. For example, a UML message sequence chart may describe a method-calling sequence. This approach is called model-based testing (MBT). The UML model is designed using a CASE tool and is then imported to the test generator. The test generator generates test scripts, which are then passed on to a suitable test execution tool.

*Test designer's creativity  
cannot be replaced*

Test tools cannot work miracles. Specifying tests is a very challenging task, requiring not only a comprehensive understanding of the test object but also creativity and intuition. A test data generator can apply certain rules (e.g., boundary value analysis) for systematic test generation. However, it cannot judge whether the generated test cases are suitable, important, or irrelevant. The test designer must still perform this creative analytical task. The corresponding expected result must be determined and added manually too.

### 7.1.3 Tools for Static Testing

Static analysis can be executed on source code or on specifications before there are executable programs. Tools for static testing can therefore be helpful to find faults in early phases of the development cycle (i.e., the left branch of the generic V-model in figure 3-1). Faults can be detected and fixed soon after being introduced and thus dynamic testing will be less riddled with problems, which decreases costs and development time.

*Tools for review support*

Reviews are structured manual examinations using the principle that four eyes find more defects than two (see section 4.1.2). Review support tools help to plan, execute, and evaluate reviews. They store information about planned and executed review meetings, meeting participants, and findings and their resolution and results. Even review aids like checklists can be provided online and maintained. The collected data from many

reviews may be evaluated and compared. This helps to better estimate review resources and to plan reviews but also to uncover typical weaknesses in the development process and specifically prevent them.

Tools for review support are especially useful when large, geographically distributed projects use several teams. Online reviews can be useful here and even may be the only possibility.

Static analyzers provide measures of miscellaneous characteristics of the program code, such as the cyclomatic number and other code metrics (see section 4.2). Such data are useful for identifying complex areas in the code, which tend to be defect prone and risky and should thus be reviewed. These tools can also check that safety- and security-related coding requirements have been followed. Finally, they can identify portability issues in the code.

*Static analysis*

Additionally, static analyzers can be used to find inconsistencies and defects in the source code. These are, for example, data flow and control flow anomalies, violation of programming standards, and broken or invalid links in website code.

Analyzers list all “suspicious” areas, whether there are really problems or not, causing the output lists to grow very long. Therefore, most tools are configurable; that is, it is possible to control the breadth and depth of analysis. When analyzing for the first time, the tools should be set to be less thorough. A more thorough analysis may be done later. In order to make such tools acceptable for developers, it is essential to configure them according to project-specific needs.

Source code is not the only thing that may be analyzed for certain characteristics. Even a specification can be analyzed if it is written in a formal notation or if it is a formal model. The corresponding analysis tools are called *model checkers*. They “read” the model structure and check different static characteristics of these models. During checking, they may find problems such as missing states, state transitions, and other inconsistencies in a model. The specification-based test generators discussed in section 7.1.2 are often extensions of static model checkers. Such tools are very interesting for developers if they generate test cases.

*Model checker*

#### 7.1.4 Tools for Dynamic Testing

When talking about test tools in general, we often mean tools for automatic execution of dynamic tests. They reduce the mechanical work involved in test execution. Such tools send input data to the test object, record its reaction, and document test execution. In most cases the tools run on the same

*Tools reduce mechanical test work*

hardware as the test object. However, this influences the test object because the test tool uses memory and machine resources. The test object may thus react differently. This is called the *tool effect*. This must be remembered when evaluating the test. These tools must be coupled to the test interface of the test object. They are therefore quite different, depending on the test level in which they are used.

#### *Debuggers*

Debuggers make it possible to execute a program or part of a program line by line. They also support stopping the program at every statement to set and read variables. Debuggers are mainly analysis tools for the developers, used to reproduce failures and analyze their causes. But even during testing, debuggers can be useful to force certain test situations, which normally requires a lot of work. Debuggers can also be used as a test interface at the component level.

#### *Test drivers and test frameworks*

Products or tailor-made tools that interact with the test object through its programming interface are called test drivers or test frameworks. This is especially important for test objects without a user interface, that is, impossible to test manually. Test drivers are mainly necessary in component and integration testing and for special tasks in system testing. Generic test drivers or test harness generators are also available. These analyze the programming interface of the test object and generate a test frame. Test framework generators are thus made for specific programming languages and development environments. The generated test frame contains the necessary initializations and call sequences to control the test object. Even dummies or stubs may be generated. Additionally, functions are provided to capture test execution and expected and actual results. Test frame (generators) considerably reduce the work necessary for programming the test environment. Different generic test frameworks are freely available on the Internet [URL: xunit].

#### *Simulators*

If performing a system test in its operational environment or using the final system is not possible or demands a disproportionately great effort (e.g., airplane control robustness test in the airplane itself), *simulators* can be used. A simulator simulates the actual application environment as comprehensively and realistically as possible.

#### *Test robots*

Should the user interface of a software system directly serve as the test interface, so-called test robots can be used. These tools have traditionally been called  $\rightarrow$ *capture/replay tools* or  $\rightarrow$ *capture/playback tools*, which almost completely explains their way of functioning. A test robot works somewhat like a video recorder: The tool logs all manual tester inputs (keyboard inputs and mouse clicks). These inputs are then saved as a test script. The

tester can repeat the test script automatically by “playing it back.” This principle sounds very tempting and easy, but in practice, there are traps.

In capture mode, the capture/playback tool logs keyboard input and mouse clicks. The tool not only records the x-/y-coordinates of the mouse clicks but also the events (e.g., `pressButton` (“Start”)) triggered in the graphical user interface (GUI) as well as the object’s attributes (object name, color, text, x/y position, etc.) necessary to recognize the selected object.

To determine if the program under test is performing correctly, the tester can include checkpoints, that is, comparisons between expected and actual results (either during test recording or during script editing).

Thus, layout properties of user interface controls (e.g., color, position, and button size) can be verified as well as functional properties of the test object (value of a screen field, contents of a message box, output values and texts, etc.).

The captured test scripts can be replayed and therefore, in principle, be repeated as often as desired. Should a discrepancy in values occur at a checkpoint, “the test fails.” The test robot then writes an appropriate notice in the test log. Because of their capability to perform automated comparisons of actual and expected values, test robot tools are extraordinarily well suited for regression test automation.

However, one problem exists: If, in the course of program correction or program extension, the test object’s GUI is changed between two test runs, the original script may not “suit” the new GUI layout. The script, no longer being synchronized to the application, may stop and abort the automated test run. Test robot tools offer a certain robustness against such GUI layout changes because they recognize the object itself and its properties, instead of just x/y positions on the screen. This is why, for example, during replay of the test script, buttons will be recognized again, even if their position has moved.

Test scripts are usually written in scripting languages. These scripting languages are similar to common programming languages (BASIC, C, and Java) and offer their well-known general language properties (decisions, loops, procedure calls, etc.). With these properties it is possible to implement even complex test runs or to edit and enhance captured scripts. In practice, this editing of captured scripts is nearly always necessary because capturing usually does not deliver scripts capable for regression testing. The following example illustrates this.

The tester will test the VSR subsystem for contract management by examining whether sales contracts are properly filed and retrieved. For test automation purposes, the tester may record the following interaction sequence:

```
Call screen "contract data";
Enter data for customer "Miller";
Set checkpoint;
Store "Miller" contract in contract database;
Clear screen "contract data";
Read "Miller" contract from contract database;
Compare checkpoint with screen contents;
```

**Excursion: On the functioning of capture/playback tools**

*Capture*

*Result comparisons*

*Replay*

*Problem: GUI changes*

*Test programming*

**Example:**  
**Automated test of VSR-ContractBase**

A successful check indicates that the contract read from the database corresponds to the contract previously filed, which leads to the conclusion that the system correctly files contracts. But, when replaying this script, the tester is surprised to find that the script has stopped unexpectedly. What happened?

*Problem:*  
*Regression test capability*

When the script is played a second time, upon trying to store the “Miller” contract, the test object reacts in a different way than during the first run. The “Miller” contract already exists in the contract database, and the test object ends the attempt to file the contract for the second time by reporting as follows:

```
"Contract already exists.  
Overwrite the contract Y/N?"
```

The test object now expects keyboard input. Because keyboard input is missing in the captured test script, the automated test stops.

The two test runs have different preconditions. Because the captured script relies on a certain precondition (“Miller” contract not in the database), the test case is not good enough for regression test. This problem can be corrected by programming a case decision or by deleting the contract from the database as the final “cleanup action” of the test case.

---

As seen in the example, it is crucial to edit the scripts by programming. Thus, implementing such test automation requires programming skills. When comprehensive and long-lived automation is required, a well-founded test architecture must be chosen; that is, the test scripts must be modularized.

***Excursion:***  
***Test automation***  
***architectures***

A good structure for the test scripts helps to minimize the cost for creating and maintaining automated tests. A good structure also supports dividing the work between test automators (knowing the test tool) and testers (knowing the application/business domain).

*Data-driven testing*

Often, a test procedure (test script) shall be repeated many times with different data. In the previous example, not only the contract of Mr. Miller should be stored and managed, but the contracts of many other customers as well.

An obvious step to structure the test script and minimize the effort is to separate test data and test procedure. Usually the test data are exported into a table or spreadsheet file. Naturally, expected results must also be stored. The test script reads a test data line, executes the →*test procedure* with these test data, and repeats this process with the next test data line. If additional test data are necessary, they are just added to the test data table without changing the script. Even testers without programming skills can extend these tests and maintain them to a certain degree. This approach is called *data-driven testing*.

*Command- or keyword-*  
*driven testing*

In extensive test automation projects, an extra requirement is reusing certain test procedures. For example, if contract handling should be tested, not only for buying new cars but also for buying used cars, it would be useful to run the script from the

previous example without changes for both areas. Thus, the test steps are encapsulated in a procedure named, for example, `check_contract` (customer). The procedure can then be called via its name and reused in other tests without changes.

With correct granularity and correspondingly well-chosen test procedure names, it is possible to achieve a situation where every execution sequence available for the system user is mapped to such a procedure or command. So that the procedures can be used without programmer know-how, an architecture is implemented to make the procedures callable through spreadsheet tables. The (business) tester will then (analogous to the data-driven test) work only with tables of commands or keywords and test data. Specialized test automation programmers have to implement the commands. This approach is called command-, keyword-, or action-word-driven testing.

The spreadsheet-based approach is only partly scalable. With large lists of keywords and complex test runs, the tables become incomprehensible. Dependencies between commands, and between commands and their parameters, are very difficult to trace. The effort to maintain the tables grows disproportionately as the tables grow.

The newest generation of test tools (for an example, see [URL: TestBench]) implement an object-oriented management of test modules using a database. You can retrieve test modules (so-called *interactions*) from the database by dragging and dropping them into new test sequences. The necessary test data (even complex data structures) are automatically included. If any module is changed, every area using this module is easy to find and can be selected. This considerably reduces the test maintenance effort. Even very large repositories can be used efficiently and without losing overview.

*Interaction-driven tests*

→Comparators (a further tool class) are used to identify differences between expected and actual results. Comparators typically function with standard file and database formats, detecting differences between data files containing expected and actual data. Test robots usually include integrated comparator functions operating with terminal contents, GUI objects, or screen content copies. These tools usually offer filtering mechanisms that skip data or data fields that are irrelevant to the comparison. For example, this is necessary when date/time information is contained in the test object's file or screen output. Because this information differs from test run to test run, the comparator would wrongly interpret this change as a difference between expected and actual outcome.

*Comparators*

During test execution, tools for dynamic analysis acquire additional information on the internal state of the software being tested. This may be, for instance, information on allocation, usage, and release of memory. Thus, memory leaks, wrong pointer allocation, or pointer arithmetic problems can be detected.

*Dynamic analysis*

Coverage analyzers provide structural test coverage values that are measured during test execution (see section 5.2). For this purpose, prior to

*Coverage analysis*

execution, an instrumentation component of the tool inserts measurement code into the test object (instrumentation). If such measurement code is executed during a test run, the corresponding program fragment is logged as “covered.” After test execution, the coverage log is analyzed and a coverage statistic is created. Most tools provide simple coverage metrics, such as statement coverage and branch coverage (see sections 5.2.1 and 5.2.2). When interpreting measurement results, it is important to bear in mind that different coverage tools may give different coverage results and that some coverage metrics may be defined differently depending on the actual tool.

### 7.1.5 Tools for Nonfunctional Test

#### *Load and performance test*

There is tool support for nonfunctional tests, especially for load and performance tests. Load test tools generate a synthetic load (i.e., parallel database queries, user transactions, or network traffic). They are used for executing volume, stress, or performance tests. Tools for performance tests measure and log the response time behavior of the system being tested depending on the load input. Depending on how the measurement is performed and on the tools being used, the time behavior of the test object may vary (probe effect or tool effect). This must be remembered when interpreting the measurement results. In order to successfully use such tools and evaluate the test results, experience with performance tests is crucial. The necessary measurement elements are called *monitors*.

#### *Monitors*

Load or performance tests are necessary when a software system has to execute a large number of parallel requests or transactions within a certain maximum response time. Real-time systems and, normally, client/server systems as well as web-based applications must fulfill such requirements. Performance tests can measure the increase in response time correlated to increasing load (for example, increasing number of users) as well as the system’s maximum capacity when the increased load leads to unacceptable latency due to overload. Used as an analysis resource, performance test tools generally supply the tester with extensive charts, reports, and diagrams representing the system’s response time and transaction behavior relative to the load applied as well as information on performance bottlenecks. Should the performance test indicate that overload already occurs under everyday load conditions, the system must be tuned (for example, by hardware extension or optimization of performance-critical software components).

Tools for testing access control and data security try to detect security vulnerabilities. Exploiting these, unauthorized persons may possibly get access to the system. Virus scanners and firewalls can also be seen as part of this tool category, mainly because the protocols generated by such tools deliver evidence of security deficiencies.

*Testing of security*

In system testing, especially in data conversion or migration projects, tools may be used to check the data. Before and after conversion or migration, the data must be checked to make sure they are correct and complete or conform to certain syntactical or semantic rules.

*Data Quality Assessment*

---

Various aspects of data quality are relevant to the VSR-System:

- The *DreamCar* subsystem includes various car model and accessory variants. Even if the software works correctly, the user may experience problems if their data concerning specific cars or accessories is incorrect or missing. In this case, the user cannot successfully create a configuration or, worse still, is able to create a configuration that is impossible to produce. The customer happily expects his new car and later learns that it cannot be delivered. Such situations guarantee customer disappointment.
- *NoRisk* helps the sales office calculate matching insurance costs for any car. If some of the data is obsolete, the cost of insurance will be wrongly priced. This is a risk, especially if the resulting quote is too expensive, which would encourage customers to seek an alternative insurance provider. The customer could decide not to sign an insurance contract at the car dealer and instead look for insurance at home through the Internet.
- *ContractBase* is used to document complete customer histories, including contracts, repairs, and so on. The prices can be shown in euros, even if they were actually transacted in an older European currency. When older currencies were converted to euros, were the data correctly converted or has the customer really paid the shown amount?
- During advertising campaigns, the car dealer regularly sends out special offers and invites existing customers to attend new product presentations. The system contains all necessary address data as well as further data, such as, for example, the age of the customer's current car. However, the special advertising will get to the right customer with the right information only if the data are complete and consistent. Does the VSR-System prevent such problems at input time? For example, does the zip code match the actual town or road? Does the system assure that all fields relevant for marketing are filled in (for example, the age of the used car and not the contract date for its purchase)?

**Example:**  
**Data quality in  
the VSR-System**

---

These examples illustrate that data accuracy is largely the responsibility of the customer or system user. But the system supplier can support "good"

data quality. Conversion programs need to be bug free and traceable. There should be reasonable checks of input data for consistency and plausibility and many other similar measures.

## 7.2 Selection and Introduction of Test Tools

Some elementary tools (e.g., comparators, coverage analyzers) are included in typical operating system environments (e.g., UNIX) as a standard feature. In these cases, the tester can assemble the necessary tool support using simple, available means. Naturally, the capabilities of such standard tools are limited, so it is sometimes useful to buy more advanced commercial test tools.

As described earlier, special tools are commercially available for every phase in the test process, supporting the tester in executing the phase-specific tasks or performing these tasks themselves. The tools range from test planning and test specification tools, supporting the tester in his creative test development process, to test drivers and test robots able to automate the mechanical test execution tasks.

When contemplating the acquisition of test tools, automation tools for test execution should not be the first and only choice.

*“Automating ... faster  
chaos”*

The area in which tool support may be advantageous strongly depends on the respective project environment and the maturity level of the development and test process. Test execution automation is not a very good idea in a chaotic project environment, where “programming on the fly” is common practice, documentation does not exist or is inconsistent, and tests are performed in an unstructured way (if at all). A tool can never replace a nonexistent process or compensate for a sloppy procedure. “It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing. Automating chaos just gives faster chaos” [Fewster 99, p. 11].

In those situations, manual testing must first be organized. This means, initially, that a systematic test process must be defined, introduced, and adhered to. Next, thought can be given to the question, Which process steps can be supported by tools? What can be done to enhance the productivity or quality by using tools? When introducing testing tools from the different explained categories, it is recommended to adhere to the following order of introduction:

1. Incident management
2. Configuration management
3. Test planning
4. Test execution
5. Test specification

*Observe the order  
of tool introduction*

Take into account the necessary time to learn a new tool and to establish its use. Due to the learning curve, productivity may even decline for some time instead of increasing, as would be desired. It is therefore risky to introduce a new tool during “hot” project phases, hoping to solve personnel bottlenecks then and there by introducing a misunderstood kind of automation.

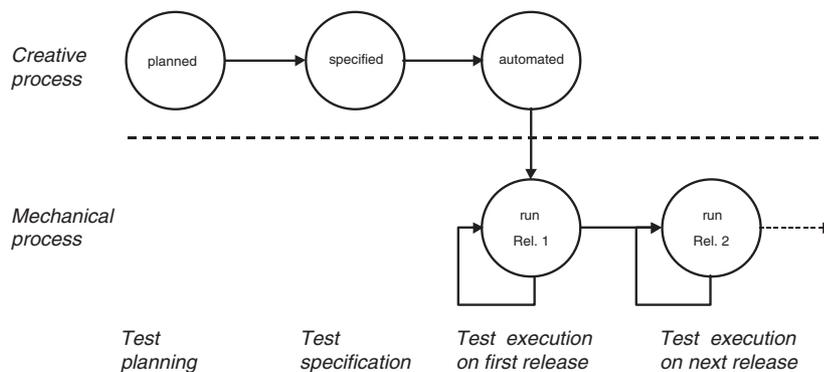
*Remember the learning  
curve*

### 7.2.1 Cost Effectiveness of Tool Introduction

Introducing a new tool brings with it selection, acquisition, and maintenance costs. In addition, costs may arise for hardware acquisition or updates and employee training. Depending on tool complexity and the number of workstations to be equipped with the tool, the amount invested can rapidly grow large. The time frame in which the new test tool will start to pay back is interesting, as with every investment.

Test execution automation tools offer a good possibility for estimating the amount of effort saved when comparing an automated test to a manually executed test. The extra test programming effort must, of course, be taken into account. This typically results in a negative cost-benefit balance after only one automated test run. The achieved savings per test run accumulate only after further automated regression test runs (figure 7–2).

*Make a cost-benefit analysis*



**Figure 7–2**  
*The life cycle of  
a test case*

The balance will become positive after a certain number of regression test cycles. It is difficult to give an exact estimate of the time for payback. Breakeven will be achieved only if the tests are designed and programmed for easy use in regression testing and easy maintenance. If tests are easy to repeat and maintain, a positive balance is definitely possible from the third test cycle onward for capture/replay tools. Of course, this calculation makes sense only when manual execution is possible at all. However, many tests cannot be run in a purely manual way (e.g., performance tests). They *have to* be run automatically.

*Evaluate the influence  
on test quality*

Merely discussing the level of test effort does not suffice. Test quality improvement by applying a new test tool that results in detecting and eliminating more faults, or results in more trustworthiness of the test, must also be taken into account. Development, support, and maintenance costs will decrease, at least in the medium term. The savings potential is significantly higher for this case and therefore more interesting.

To summarize:

- Tools may support the creative testing tasks. They help the tester to improve test quality.
- Mechanical test execution may be automated. This reduces test effort or makes it possible to run more tests with the same test effort. However, more tests do not necessarily mean better test quality.
- In both cases, without good test procedures or well-established test methods, tools do not lead to the desired cost reduction.

### **7.2.2 Tool Selection**

After a decision is made about which test task a tool shall support, the actual selection (and evaluation) of the tool starts. As explained earlier, the investment can be very large. It is, therefore, best to proceed carefully and in a well-planned way. The selection process consists of the following five steps:

1. Requirements specification for the tool
2. Market research (creating a list of possible candidates)
3. Tool demonstrations
4. Evaluation of the tools on the short list
5. Review of the results and selection of the tool

For the first step, requirements specification, the following criteria may be relevant:

- Quality of interaction with the potential test objects
- Tester know-how regarding the tool or method
- Possibility of integration into the existing development environment
- Possibility of integration with other already used testing tools
- Platform for using the tool
- Possibilities for integration with tools from the same supplier
- Manufacturer's service, reliability, and market position
- License conditions, price, maintenance costs

*Selection criteria*

These and possible further individual criteria are compiled in a list and then weighted according to their relative importance. Absolutely indispensable criteria are identified and marked as knock-out criteria.

Parallel to creating a catalogue of criteria, market research takes place: A list is created, listing the available products of the tool category of interest. Product information is requested from suppliers or collected from the Internet. Based on these materials, the suppliers of the preferred candidates are invited to demonstrate their respective tools. A relatively reliable impression of the respective company and its service philosophy can be gained from these demonstrations. The best vendors will then be taken into the final evaluation process, where primarily the following points need to be verified:

*Market research  
and short-listing*

- Does the tool work with the test objects and the development environment?
- Are the features and quality characteristics that caused the respective tool to be considered for final evaluation fulfilled in reality? (Marketing can promise a lot.)
- Is the supplier's support staff able to provide qualified information and help even with nonstandard questions (before and after purchase<sup>2</sup>)?

### 7.2.3 Tool Introduction

After a tool is selected, it must be introduced into the organization. Normally the first step is to launch a pilot project (proof of concept). This should show that the expected benefits will be achieved in real projects.

---

2. Many suppliers just refer to a general hotline after purchase.

People other than the ones who helped in selecting the tool should execute the pilot project. Otherwise, the evaluation results could introduce bias.

*Pilot operation*

Pilot operation should deliver additional knowledge of the technical details of the tool as well as experiences with the practical use of the tool and experiences about its usage environment. It should thus become apparent whether, and to what extent, there exists a need for training and where, if necessary, the test process should be changed. Furthermore, rules and conventions for general use should be developed. These may be naming conventions for files and test cases, rules for structuring the tests, and so on. If test drivers or test robots are introduced, it can be determined during the pilot project if it is reasonable to build test libraries. This should facilitate reuse of certain tests and test modules outside the project.

Because the new tool will always generate additional workload in the beginning, the introduction of a tool requires strong and ongoing commitment of the new users and stakeholders.

Coaching and training measures are important.

*Success factors*

There are some important success factors during rollout:

- Introduce the tool stepwise.
- Integrate the tool's support with the processes.
- Implement user training and continuous coaching.
- Make available rules and suggestions for applying the tool.
- Collect usage experiences and make them available to all users (hints, tricks, FAQs, etc.).
- Monitor tool acceptance and gather and evaluate cost-benefit data.

Successful tool introduction follows these six steps:

1. Execute a pilot project.
2. Evaluate the pilot project experiences.
3. Adapt the processes and implement rules for usage.
4. Train the users.
5. Introduce the tool stepwise.
6. Offer coaching.

This chapter pointed out many of the difficulties and the additional effort involved when selecting and using tools for supporting the test process.

This is not meant to create the impression that using tools is not worthwhile.

---

On the contrary, in larger projects, testing without the support of appropriate tools is not feasible. However, a careful tool introduction is necessary, otherwise the expensive tool quickly becomes “shelfware”; that is, it falls into disuse.

## 7.3 Summary

- Tools are available for every phase of the test process, helping the tester automate test activities or improve the quality of these activities.
- Use of a test tool is beneficial only when the test process is defined and controlled.
- Test tool selection must be a careful and well-managed process because introducing a test tool may incur large investments.
- Information, training, and coaching must support the introduction of the selected tool. This helps to assure the future users' acceptance and hence the continued application of the tool.



---

## A Test Plan According to IEEE Standard 829-1998

This appendix describes the contents of a test plan according to IEEE Standard 829-1998.<sup>1</sup> It can be used as a guide to prepare a test plan.

### Test Plan Identifier

Specify uniquely the name and version of the test plan. The identifier must make it possible to clearly and precisely refer to this document distinct from other project documents. A standard for document identification is often given by rules set by the project archive or by the organization's central document management. Depending on the size of the project organization, the identifier may be more or less complicated. The minimum information to be used is the name of the test plan, its version, and its status.

### Introduction

The introduction should give a short summary of the project background.

Its intent is to help those involved in the project (customer, management, developer, and tester) to better understand the contents of the test plan.

A list of documents used in developing this plan or referred to should be included in this chapter. This typically includes policies and standards, such as industry standards, company standards, project standards, cus-

---

1. The latest version of the IEEE 829-2008 standard [IEEE 829-2008] differentiates between Master Test Plan and Level Test Plan. An existing test plan created according to IEEE 829-1998 can be converted to conform to an IEEE 829-2008 master test plan—for example, by using an appropriate table of cross-references. However, the 2011 syllabus still cites IEEE 829-1998 as relevant for the Certified Tester – Foundation Level examination.

tomers standards, the project authorization (possibly the contract), project plan and other plans, and the specification.

### **Test Objects or Items**

This section should contain a short overview of the parts and components of the product to be tested; a list of the test items including their version/revision level; their transmittal media and their specification. In order to avoid misunderstandings, there should be a list of system parts not subject to testing.

### **Features to Be Tested**

This section should identify all functions or characteristics of the system to be tested. The test specification should be referred to. This section should contain an assignment to test levels or test objects.

### **Features Not to Be Tested**

In order to avoid misunderstanding and prevent unrealistic expectations, it should be described which aspects of the product shall not or cannot be tested. (This may be due to resource constraints or technical reasons).

---

**Hint**

- Because the test plan is prepared early in the project, this list will be incomplete. Later it may be found that some components or features cannot be tested anyway. The test manager should then issue warnings in the status reports.
- 

### **Test Approach or Strategy**

This section should describe the test objectives, if possible, based on a risk analysis. The analysis shows which risks are threatening if faults are not found due to lack of testing. From this it can be derived which tests must be executed and which are more or less important. This assures that the test is concentrated on important topics.

Building on this, the test methods to be used are selected and described. It must be clearly visible if and why the chosen methods are able to achieve the test objectives, considering the identified risks and the available resources.

### Acceptance Criteria (Test Item Pass/Fail Criteria)

After all tests for a test object have been executed, it must be determined, based on the test results, if the test object can be released<sup>2</sup> and delivered.<sup>3</sup> Acceptance criteria or test exit criteria are defined for this end.

The criterion “defect free” is, in this context, not a very useful criterion because testing cannot prove that a product has no faults. Usually, criteria therefore include a combination of “number of tests executed,” “number of faults found,” and “severity of faults found.”

For example, at least 90% of the planned tests are executed correctly and no class 1 faults (crashes) have been found.

Such acceptance criteria can vary between the test objects. The actual definition of the criteria should be made dependent on the risk analysis; that is, for uncritical test objects, acceptance criteria can be weaker than for safety-critical test objects, for example. Thus, the test resources are concentrated on important system parts.

### Suspension Criteria and Resumption Requirements

Aside from acceptance criteria, there is also a need for criteria to indicate a suspension or cancellation of the tests.

It may be that a test object is in such bad shape that there is no chance it will be accepted, even after an enormous amount of testing. To avoid such wasteful testing, we need criteria that will lead to termination of useless testing early in the testing life cycle. The test object will then be returned to the developer without executing all tests.

Analogous to this, criteria for resumption or continuation of the tests are needed. The responsible testers will typically execute an entry test (smoke test). Only after this is executed without trouble should the real test begin.

- 
- Criteria should involve only measurements that can be measured regularly, easily, and reliably—for example, because they are automatically collected by the test tools used. The test manager should then list and interpret this data in every test report.
- 

**Hint**

---

2. Release is a management decision that the tested product is seen as “ready.”  
3. To deliver may also mean to send the test object to the next test level.

### **Test Documentation and Deliverables**

In this section, we describe which data and results the test activities will deliver and in which form and to whom these results will be communicated. This not only means the test results in a narrow sense (for example, incident reports and test protocols), it also includes planning and preparation documents such as test plans, test specifications, schedules, documents describing the transmittal of test objects, and test summary reports.

- 
- Hint** ■ In a test plan, only formal documentation is mentioned. However, informal communication should not be forgotten. Especially in projects that are already in trouble, or in very stressful phases (for example, the release week), an experienced test manager should try to directly communicate with the involved people more than he usually would. This is not to conceal bad news, but it should be used to assure that the right consequences are chosen after possible bad news.
- 

### **Testing Tasks**

This section contains a list of all tasks necessary for the planning and execution of the tests, including an assignment of responsibilities. The status of the tasks (open, in progress, delayed, done) must be followed up. This point in the test plan is part of the normal project planning and follow-up and should therefore be reported in the regular project or test status reports, which are referred to here.

### **Test Infrastructure and Environmental Needs**

This section lists the elements of the test infrastructure that are necessary to execute the planned tests. This typically includes test platform(s), tester workplaces and their equipment, test tools, development environment (whatever is necessary for the testers), and other tools (email, Internet access, Office software packages, etc.).

- 
- Hint** ■ The test manager should consider the following aspects: Acquisition of the unavailable parts of a previously mentioned “wish list”; questions about budget, administration, and operation of the test infrastructure; the test objects; and tools. Often, this requires specialists, at least for some time. Specialists may need to be from other departments or from external providers.
-

---

### Responsibilities and Authority

How is testing organized with respect to the project? Who has what authority, availability, etc.? Possibly the test personnel must be divided into different test groups or levels. Which people have which tasks?

- 
- Responsibilities and authority may change during the course of the project. Therefore, the list of responsibilities should be presented as a table, maybe an appendix to the test plan.
- 

*Hint*

### Staffing and Training Needs

This section specifies the staffing needs for implementing the plan (roles, qualifications, capacity, and when they are needed, as well as planning vacations, etc.). This planning is not only for the test personnel, it should also include personnel for administrating the test infrastructure (see above), as well as developers and customers to be included in testing.

Plans for training to provide necessary skills should be included.

### Schedule

This section describes an overall schedule for the test activities, with the major milestones. The plan must be coordinated with the project plan and maintained there. Regular consultation between the project manager and the test manager must be implemented. The test manager should be informed about delays during development and must react by changing the detailed test plan. The project manager must react to test results and, if necessary, delay milestones because extra correction and testing cycles must be executed.

- 
- The test manager must assure that the test and quality assurance activities are included in the project plan. He or she must not be an independent “state in the state.”
- 

*Hint*

### Risks and Contingencies

In the section about test approach, risks in the test object or its use are addressed. This section, however, addresses risks within the testing project itself (that is, risks when implementing the test plan, and risks resulting from not implementing wanted activities) because it was already clear

when planning that there would be no resources for them in the concrete project.

The minimum should be a list of risks to be monitored at certain points in time (for example, in connection with the regular test status reports) in order to find measures to minimize them.

### **Approval**

This section contains a list of the persons or organizational units that will approve the test plan or need to be informed. Approval should be documented by signature. It should also be documented that parties have been informed after major changes, especially changes of strategy and/or personnel.

---

#### **Hint**

- Respective groups or organizational units are typically development group(s), project management, project steering committee, software operators, software users, customers (clients) and naturally, the test group(s).
- 

Depending on the project situation, the intent of the approval described here may be different.

The ideal situation is, “You approve that the mentioned resources will be financed and used in order to test this system as described here in a reasonable way.”

The often-more-usual situation is, “Because of the lack of resources shown, tests can only be performed in an insufficient manner, testing the very most important parts. However, you approve this and accept that based on this, release decisions will be made, which may have a high risk.”

### **Glossary (not in IEEE829-1998, but lower case!)**

Testing has no tradition for using standardized terminology. Thus, the test plan should contain an explanation of the testing terms used in the project. There is a high danger that different people will have different interpretations of testing terms. For example, just ask several people involved in the project for the definition of the term *load testing*.

## Test Plans According to IEEE Standard 829-2008

This appendix describes the contents of a test plan according to IEEE Standard 829-2008. It can be used as a guide to prepare a test plan.

This version of the IEEE 829-2008 standard [IEEE 829-2008] differentiates between →Master Test Plan and →Level Test Plan. An existing test plan created according to IEEE 829-1998 can be converted to conform to an IEEE 829-2008 master test plan and level test plans—for example, by using an appropriate table of cross-references. The 2015 syllabus will use IEEE 829-2008 as reference for the Certified Tester – Foundation Level examination.

### The Master Test Plan

The objective of a master test plan is to describe the overall test approach in a project or an organization. It describes all the different test levels, test activities, and test tasks to be done and their relationship. Level test plans, on the other hand, describe what is to be done in one test level. The older IEEE 829 standard had only one test plan, and it was not clear if it applied to a whole project or one level.

---

### The template from the standard<sup>4</sup>

#### 1. Introduction

- 1.1. Document identifier
- 1.2. Scope
- 1.3. References
- 1.4. System overview and key features
- 1.5. Test overview
  - 1.5.1 Organization
  - 1.5.2 Master test schedule
  - 1.5.3 Integrity level schema
  - 1.5.4 Resources summary
  - 1.5.5 Responsibilities
  - 1.5.6 Tools, techniques, methods, and metrics

---

4. The standard's section 2.1. actually contains more details. In order to shorten it and ease the overview, only details for the development process are given here.

## **2. Details of the Master Test Plan**

- 2.1. Test processes including definition of test levels
  - 2.1.1 Process: Management
  - 2.1.2 Process: Acquisition
  - 2.1.3 Process: Supply
  - 2.1.4 Process: Development
    - 2.1.4.1 Activity: Concept
    - 2.1.4.2 Activity: Requirements
    - 2.1.4.3 Activity: Design
    - 2.1.4.4 Activity: Implementation
    - 2.1.4.5 Activity: Test
    - 2.1.4.6 Activity: Installation/checkout
  - 2.1.5 Process: Operation
  - 2.1.6 Process: Maintenance
    - 2.1.6.1 Activity: Maintenance test
- 2.2. Test documentation requirements
- 2.3. Test administration requirements
- 2.4. Test reporting requirements

## **3. General**

- 3.1. Glossary
- 3.2. Document change procedures and history

### **Master Test Plan Identifier**

Specify the name and version of the test plan. The identifier must make it possible to clearly and precisely refer to this document distinct from other project documents. The minimum information to be used is the name of the test plan, its version, and its status.

### **1. Introduction**

The introduction should give a short summary of the project background.

Its intent is to help the readers of the plan (customer, management, developer, regulating authorities, and tester) to better understand the contents of the test plan.

The introduction should describe the entire test effort, including the test organization, the test schedule, and the integrity schema. A summary of required resources, tools and techniques might also be included in this section.

## **1.2 Scope**

Describe the purpose, goals, and scope of the test effort.

Identify the project(s) or product(s) for which the plan is written and the specific processes and products covered by the test effort. Describe what is included and excluded, as well as assumptions and limitations. It is important to define clearly the limits of the test effort to control expectations.

## **1.3 References**

Include here a list of all of the applicable reference documents. The references are separated into “external” references that are imposed external to the project and “internal” references that are imposed from within the project. This section may also be at the end of the document, for example in chapter 3.

Referenced documents should include policies and standards, such as industry standards, company standards, project standards, customer standards, the project authorization (possibly the contract), project plan and other plans, and the specification.

## **1.4 System Overview and Key Features**

This section should present the mission or business purpose of the system or software product under test as well as a short overview of the features, parts, and components of the product to be tested; a list of the test items including their version/revision level; their transmittal media and their specification. To avoid misunderstandings, there should also be a list of system parts not subject to testing, i.e., an overall summary of the “features not to be tested” chapter in the test plan according to the older standard.

## **1.5 Test overview**

In this section, describe the test organization, the overall test schedule, and the integrity level scheme to be used to control testing, the major test resources, responsibilities, tools, techniques, and methods to be applied.

### **1.5.1 Organization**

Describe the relationship of the test processes to other development and supporting processes. It may be beneficial to include an organization chart. Describe how testing and other tasks shall communicate.

### 1.5.2 Master Test Schedule

Outline an overall schedule for the test activities, with the major milestones. The test plan must be coordinated with the project plan and maintained throughout the project. Regular consultation between the project manager and the test manager must be implemented. The test manager should be informed about delays during development and must react by changing the test plan. The project manager must react to test results and, if necessary, delay milestones because extra correction and testing cycles must be executed.

To handle changes and iterations, describe the task iteration policy for the re-execution of test tasks and any dependencies.

### 1.5.3 Integrity Level Scheme

Describe how integrity levels are identified and how they govern the testing effort. The plan should document the assignment of integrity levels to individual documents and components as well as how integrity levels are used to control the testing tasks.

### 1.5.4 Summary of Necessary Resources

Describe the needed test resources, including staffing, facilities, tools, and special procedural requirements like security, access rights, and documentation control. Include a description of training needs.

### 1.5.5 Responsibilities and Authority

How is testing organized with respect to the organization and the project? Who has what authority, availability, etc.? Possibly the test personnel must be divided into different test groups or levels. Which people have which tasks? Who shall provide support to testing?

---

**Hint**

- Responsibilities and authority may change during the course of the project. Therefore, the list of responsibilities should be presented as a table, maybe an appendix to the test plan.
- 

### 1.5.6 Tools, Techniques, Methods, and Metrics

Describe documents, hardware and software, test tools, techniques, methods, and test environment necessary for the test process. Describe the techniques to be used to identify and capture reusable testware (for regression

testing). Include information regarding acquisition, training, support, and qualification for each tool, technology, and method, at least for everything new to the organization.

Document the metrics to be used by the test effort, and describe how these metrics will be collected, evaluated, and used to support the test objectives.

More details about topics regarding specific test levels may be included in level test plans.

## **2. Details of the Master Test Plan**

This section describes the test processes, test documentation requirements, and test reporting requirements for the entire test effort.

### **2.1 Test Processes, Including Definition of Test Levels**

Identify test activities and tasks to be performed for each of the test processes and document those test activities and tasks. Provide an overview of the test activities and tasks for all development life cycle processes. Identify the test levels, including any “special” tests like security, usability, performance, stress, recovery, and regression testing.

If the test processes are already defined by an organization’s standards, a reference to those standards could be substituted for the contents of this section.

#### **2.1.1 through 2.1.6 “Life cycle” Processes, i.e., Activities and Tasks**

There may be up to six subsections here, for the life cycle processes Management, Acquisition, Supply, Development, Operation, and Maintenance. Normally for a development project, there is only the subsection about development.

This section contains a list of all activities and tasks necessary for the planning and execution of the tests, including an assignment of responsibilities. The status of all these tasks (not started, in progress, delayed, done) must be followed up.

Address the following eight topics for each test activity<sup>5</sup>:

- a) *Test tasks*: Identify the test tasks to be performed as well as the degree of intensity and rigor in performing and documenting the task.

---

5. These eight topics are cited from IEEE 829-2008.

- b) *Methods*: Describe the methods and procedures for each test task, including tools. Define the criteria for evaluating the test task results.
- c) *Inputs*: Identify the required inputs for the test task. Specify the source of each input. For any test activity and task, any of the inputs or outputs of the preceding activities and tasks may be used.
- d) *Outputs*: Identify the required outputs from the test task. The outputs of the management of the test and of the test tasks will become inputs to subsequent processes and activities, as appropriate.
- e) *Schedule*: Describe the schedule for the test tasks. Establish specific milestones for initiating and completing each task, for the receipt of each input, and for the delivery of each output.
- f) *Resources*: Identify the resources for the performance of the test tasks. Specify resources by category (e.g., staffing, tools, equipment, facilities, travel budget, and training).
- g) *Risks (project risks) and Assumptions*: Identify the risk(s) (e.g., schedule, resources, technical approach, or for going into production) and assumptions associated with the test tasks. Provide recommendations to eliminate, reduce, or mitigate risk(s). This section takes much of the information provided in the section “Risks and contingencies” of the old standard.
- h) *Roles and responsibilities*: Identify for each test task the organizational elements that have the responsibilities for the execution of the task, and the nature of the roles they will play.

## 2.2 Test Documentation Requirements

In this section, we describe which data and results the test activities will deliver and in which form and to whom these results will be communicated. This not only means the test results in a narrow sense (for example, incident reports and test protocols), it also includes planning and preparation documents such as test plans, test specifications, schedules, documents describing the transmittal of test objects, and test summary reports.

- 
- Hint** ■ In a test plan, only formal documentation is mentioned. However, informal communication should not be forgotten. Especially in projects that are already in trouble, or in very stressful phases (for example, the release week), an experienced test manager should try to directly communicate with the involved

people more than he usually would. This is not to conceal bad news, but it should be used to assure that the right consequences are chosen after possible bad news.

---

### **2.3 Test Administration Requirements**

This section should describe how the test will be administered in practice, during its execution.

#### **2.3.1 Anomaly (defect) Resolution and Reporting**

Describe the method of reporting and resolving anomalies (incidents, failures). This section of the plan defines the criticality levels for defects. Classification for software anomalies may be found in chapter 6 of this book. This section may also refer to a general standard way of defect handling in the organization.

#### **2.3.2 Task Iteration Policy**

Describe the criteria used to determine the extent to which a testing task is repeated after changes (e.g., re-reviewing, retesting, and regression testing after problems have been repaired). These criteria may include assessments of change extent and risk, integrity level, and effects on budget, schedule, and quality.

#### **2.3.3 Deviation Policy**

Describe the procedures and criteria used to deviate from the master test plan and level test plans. Identify the authorities responsible for approving deviations.

#### **2.3.4 Control Procedures**

Identify control procedures applied to the test activities. These procedures describe how the system and software products and test results will be configured, protected, and stored.

These procedures may describe quality assurance, configuration management, data management, or other activities if they are not addressed in other plans or activities. Describe any security measures necessary for the test effort.

### **2.3.5 Standards, Practices, and Conventions**

Identify or reference the standards, practices, and conventions that govern testing tasks, if they are not “matters of fact.”

### **2.4 Test Reporting Requirements**

Specify the purpose, content, format, recipients, and timing of all test reports. Test reporting consists of Test Logs, Anomaly (failure, incident) Reports, Interim Test Status Report(s), Level Test Report(s), and the Master (or final) Test Report. Test reporting may also include other reports as deemed necessary.

## **3. General**

This section includes general information and could as well be put into chapter 1 or at the title page or into a general place accessible for all people in the project.

### **3.1 Glossary**

Testing has no tradition for using standardized terminology. Thus, the test plan should contain an explanation of the testing terms used in the project. There is a high danger that different people will have different interpretations of testing terms. For example, just ask several people involved in the project for the definition of the term *load testing*.

Thus: Provide an alphabetical list of terms and acronyms that may require definition for the users of the plan with their corresponding definitions. You may also refer to a project glossary.

### **3.2 Document Change Procedures and History**

The section should define the configuration management procedures to be followed for this document, if they are different from other documents. But at least the change list and history should be included.

---

## The Level Test Plan

This is a test plan for only one test level, like acceptance test plan, system test plan, etc.

### The template from the standard

#### 1. Introduction

- 1.1. Document identifier
- 1.2. Scope
- 1.3. References
- 1.4. Level in the overall sequence
- 1.5. Test classes and overall test conditions

#### 2. Details for this level of test plan

- 2.1 Test items and their identifiers
- 2.2 Test Traceability Matrix
- 2.3 Features to be tested
- 2.4 Features not to be tested
- 2.5 Approach
- 2.6 Item pass/fail criteria
- 2.7 Suspension criteria and resumption requirements
- 2.8 Test deliverables

#### 3. Test management

- 3.1 Planned activities and tasks; test progression
- 3.2 Environment/infrastructure
- 3.3 Responsibilities and authority
- 3.4 Interfaces among the parties involved
- 3.5 Resources and their allocation
- 3.6 Training
- 3.7 Schedules, estimates, and costs
- 3.8 Risk(s) and contingency(s)

#### 4. General

- 4.1 Quality assurance procedures
- 4.2 Metrics
- 4.3 Test coverage
- 4.4 Glossary
- 4.5 Document change procedures and history

It can be seen that chapters 2 and 3 in a level test plan contain many of the points the old standard test plan contained. Guidance for these sections is given before in this Appendix, under the heading of the old standard (829-1998). Section 3.4 is new in this standard. New are also sections 4.1 through 4.3. Sections 4.4 and 4.5 may be placed the same way as in the master test plan.

### **Guidance for section 3.4**

Describe the communication between the individuals and groups identified in section 3.5. This includes what needs to be communicated, how, and when. A figure that illustrates the flow of information and data may be included.

### **Guidance for sections 4.1 through 4.3**

#### **4.1 Quality assurance procedures**

Identify the means by which the quality of testing processes and products will be assured. Include or reference procedures for how problems in testing will be tracked and resolved. A general Quality Assurance Plan or Standard Procedure may be referenced, if there exists one.

#### **4.2 Metrics**

Identify the specific measures that will be collected, analyzed, and reported. The metrics specified here are those that only apply to this particular test level (the global metrics are described in Master Test Plan section 1.5.6). This may be a reference to where metrics are documented in a Quality Assurance Plan or to a generally used measurement program.

#### **4.3 Test coverage**

Specify the requirement(s) for test coverage. The type of coverage that is relevant varies by the level of test. See chapter 5 in this book for details. For example, unit test coverage is often expressed in terms of percentage of code tested (white box test coverage), and system test coverage can be a percentage of requirements tested (black box test coverage).

---

## **B Important Information about the Syllabus and the Certified Tester Exam**

The Certified Tester Foundation Level syllabus version 2011 forms the basis of this book. A few updates to the syllabus, which is due to be released in 2015, are noted in the book.

The respective national boards may create and maintain additional national versions of the syllabus. These may contain minor deviations from the English original, such as, for example, references to local standards. The national boards coordinate and guarantee mutual compatibility of their curricula and exams. In this context, the responsible board is the International Software Testing Qualifications Board [URL: ISTQB].

The exams are based on the current version of the syllabus in its corresponding examination language at the time of examination. The exams are offered and executed by the respective national board or by the appointed certification body. Further information on the curricula and the exams can be found through [URL: ISTQB]. The ISTQB website provides links to the national boards.

For didactic reasons, the subjects contained in this book may be addressed in a different order than presented in the syllabus. The size of the individual chapters does not indicate the relevance of their contents for the exam. Some subjects are covered in more detail in the book. Some passages, marked as excursions, go beyond the scope of the syllabus. In any case, the exams are based on the official syllabus.

ISTQB will release a new version of the Foundation Level syllabus in 2015. Some changes that were obvious at the time this book was written are included and noted as such.

The exercises and questions contained in this book should be regarded solely as practice material and examples. They are not representative of the official examination questions.

They are presented only so the reader can get a better understanding of the material.

Readers who use this book to prepare for the exam should also look at the ISTQB syllabus, official mock exams, exam rules, and the ISTQB glossary. The examination is based only on the ISTQB documents; see [URL: ISTQB] as well as your national board website.

---

## C Exercises<sup>1</sup>

### Exercises for chapter 2

- 2.1 Define the terms *failure*, *fault*, and *error*.
- 2.2 What is defect masking?
- 2.3 Explain the difference between testing and debugging.
- 2.4 Explain why each test is a kind of sampling.
- 2.5 List the main characteristics of software quality according to ISO 9126. (To prepare for the examination after syllabus version 2015, refer to ISO 25010:2011.)
- 2.6 Define the term *reliability*.
- 2.7 Explain the phases of the fundamental test process.
- 2.8 What is a test oracle?
- 2.9 Why shouldn't a developer test her own programs?

### Exercises for chapter 3

- 3.1 Explain the different phases of the general V-model.
- 3.2 Define the terms *verification* and *validation*.
- 3.3 Explain why verification makes sense even when a careful validation is performed (and vice versa).
- 3.4 Characterize typical test objects in component testing.
- 3.5 Discuss the idea of “test-first.”
- 3.6 List the goals of the integration test.
- 3.7 What integration strategies exist and how do they differ?
- 3.8 Name the reasons for executing tests in a separate test infrastructure.

---

1. These exercises are designed to help you make sure you have understood the terms and processes described in the book and that you can recognize and describe them. Answers to the questions can be found in the relevant chapters and are not listed separately. The questions listed here are not designed to directly prepare you for the ISTQB examination. To prepare for the ISTQB exam, please refer to the mock exam shown at [URL: ISTQB].

- 3.9 Describe four typical forms of acceptance tests.
- 3.10 Explain requirements-based testing.
- 3.11 Define *load test*, *performance test*, and *stress test*, and describe the differences between them.
- 3.12 How do retest and regression tests differ?
- 3.13 Why are regression tests especially important in incremental development?
- 3.14 According to the general V-model, during which project phase should the test plan be defined?

#### **Exercises for chapter 4**

- 4.1 Describe the fundamental steps for executing a review.
- 4.2 What different kinds of reviews exist?
- 4.3 Which roles participate in a technical review?
- 4.4 What makes reviews an efficient means for quality assurance?
- 4.5 Explain the term *static analysis*.
- 4.6 How are static analysis and reviews related?
- 4.7 Static analysis cannot uncover all program faults. Why?
- 4.8 What different kinds of data flow anomalies exist?

#### **Exercises for chapter 5**

- 5.1 What is a dynamic test?
- 5.2 What is the purpose of a test harness?
- 5.3 Describe the difference(s) between black box and white box test design techniques.
- 5.4 Explain the equivalence class partition technique.
- 5.5 What is a representative?
- 5.6 Define the test completeness criterion for equivalence class coverage.
- 5.7 Why is boundary value analysis a good supplement to equivalence class partitioning?
- 5.8 List further black box techniques.
- 5.9 Explain the term *statement coverage*.
- 5.10 What is the difference between statement and branch testing?
- 5.11 What is the purpose of instrumentation?

**Exercises for chapter 6**

- 6.1 What basic models can be distinguished for division of responsibility for testing tasks between development and test?
- 6.2 Discuss the benefits and drawbacks of independent testing.
- 6.3 Which roles are necessary in testing and which qualifications are necessary to fill them?
- 6.4 State the typical tasks of a test manager.
- 6.5 Discuss why test cases are prioritized and mention criteria for prioritizing.
- 6.6 What are the purposes of test start and exit criteria? Name and describe examples of such criteria.
- 6.7 Define the term *test strategy*.
- 6.8 Discuss four typical approaches to determine a test strategy.
- 6.9 Define the term *risk* and mention risk factors relevant for testing.
- 6.10 What idea is driving risk-based testing?
- 6.11 What different kinds of metrics can be distinguished for monitoring test progress?
- 6.12 What information should be contained in a test status report?
- 6.13 What data should be contained in an incident report?
- 6.14 What is the difference between defect priority and defect severity?
- 6.15 What is the purpose of an incident status model?
- 6.16 What is the task of a change control board?
- 6.17 From the point of view of testing, what are the requirements for configuration management?
- 6.18 What different kinds of basic standards exist?

**Exercises for chapter 7**

- 7.1 What main functions do test management tools offer?
- 7.2 Why is it reasonable to couple requirements and test management tools and exchange data?
- 7.3 What different types of test data generators exist?
- 7.4 What type of test data generator can also generate expected output values? Why can't other types of test data generators do the same?
- 7.5 What is a test driver?
- 7.6 Explain the general way a capture/playback tool works.
- 7.7 Describe the principle of data-driven testing.
- 7.8 What steps should be taken when selecting a test tool?
- 7.9 What steps should be taken when introducing a tool?



---

# Glossary

*This glossary contains terms from the area of software testing as well as additional software engineering terms related to software testing. The terms are marked with an arrow at their first place of occurrence in the book. The terms not listed in the ISTQB glossary are underlined in this glossary.*

*The definitions of most of the following terms are taken from the “Standard Glossary of Terms used in Software Testing” Version 2.2 (2013), produced by the Glossary Working Party of the International Software Testing Qualifications Board (ISTQB). You can find the current version of the glossary here: [URL: ISTQB]. The ISTQB glossary refers to further sources of definitions.*

## **acceptance test(ing)**

Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers, or other authorized entity to determine whether or not to accept the system.

This test may be

1. A test from the user viewpoint
2. A partial set of an existing test, which must be passed, and an entry criterion (start criterion) for the test object to be accepted into a test level

## **actual result**

The behavior produced/observed when a component or system is tested.

## **alpha testing**

Simulated or actual operational testing by potential customers/users or an independent test team at the developer’s site but outside of the development organization. Alpha testing is used for off-the-shelf software as a form of internal acceptance testing.

**analytical quality assurance**

Diagnostic based measures (for example, testing) to measure or evaluate the quality of a product.

**anomaly**

Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. [IEEE 1044] Also called *bug*, *defect*, *deviation*, *error*, *fault*, *failure*, *incident*, *problem*.

**atomic (partial) condition**

Boolean expression containing no Boolean operators (AND, OR, NOT, etc.), maximally containing relational operators like < or >.

**audit**

An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify the following:

- The form or content of the products to be produced
- The process by which the products shall be produced
- How compliance to standards or guidelines shall be measured

**back-to-back testing**

Testing in which two or more variants of a component or system are executed with the same inputs, and then the outputs are compared and analyzed in case of discrepancy.

**bespoke software**

*See special software.*

**beta testing**

Operational testing by representative users/customers in the production environment of the user/customer. With a beta test, a kind of external acceptance test is executed in order to get feedback from the market and in order to create an interest with potential customers. It is done before the final release. Beta test is often used when the number of potential production environments is large.

**black box test design techniques**

Repeatable procedure to derive and/or select test cases based on an analysis of the specification, either functional or nonfunctional, of a component or system without reference to its internal structure.

**blocked test case**

A test case that cannot be executed because the preconditions for its execution cannot be fulfilled.

**boundary value analysis**

Failure-oriented black box test design technique in which test cases are designed based on boundary values (at boundaries or right inside and outside of the boundaries of the equivalence classes).

**branch**

The expression *branch* is used as follows:

- When a component uses a conditional change of the control flow from one statement to another one (for example, in an IF statement).
- When a component uses a nonconditional change of the control flow from one statement to another one, with the exception of the next statement (for example, using GOTO statements).
- When the change of control flow is through more than one entry point of a component. An entry point is either the first statement of a component or any statement that can be directly reached from the outside of the component.

A branch corresponds to a directed connection (arrow) in the control flow graph.

**branch coverage**

The percentage of branches or decisions of a test object that have been executed by a test suite.

**branch testing**

A control-flow-based white box test design technique that requires executing all branches of the control flow graph (or every outcome of every decision) in the test object.

**bug**

See *defect*.

**business-process-based testing**

An approach to testing in which test cases are designed based on descriptions and/or knowledge of business processes.

**capture/playback tool, capture-and-replay tool**

A tool for supporting test execution. User inputs are recorded during manual testing in order to generate automated test scripts that are executable and repeatable. These tools are often used to support automated regression testing.

**cause-effect graphing**

A function-oriented black box test design technique in which test cases are designed from cause-effect graphs, a graphical form of the specification. The graph contains inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

**change**

Rewrite or new development of a released development product (document, source code).

**change order**

Order or permission to perform a change of a development product.

**change request**

1. Written request or proposal to perform a specific change for a development product or to allow it to be implemented.
2. A request to change some software artifact due to a change in requirements.

**class test**

Test of one or several classes of an object-oriented system.

See also *component testing*.

**code-based testing**

See *structural test* and *white box test design technique*.

**commercial off-the-shelf (COTS) software**

A software product that is developed for a larger market or the general market (i.e., for a large number of customers) and that is delivered to many customers in identical form. Also called *standard software*.

**comparator**

A tool to perform automated comparison of actual results with expected results.

**complete testing**

See *exhaustive testing*.

**component**

1. A minimal software item that has its own specification or that can be tested in isolation.
2. A software unit that fulfills the implementation standards of a component model (EJB, CORBA, .NET).

**component integration test(ing)**

See *integration test(ing)*.

**component test(ing)**

The testing of individual software components.

**concrete (physical or low level) test case**

A test case with concrete values for the input data and expected results.

See also *logical test case*.

**condition test(ing)**

Control-flow-based white box test design technique in which every (partial) condition of a decision must be executed both TRUE and FALSE.

**condition determination testing**

A white box test design technique in which test cases are designed to execute single-condition outcomes that independently affect a decision outcome.

**configuration**

1. The composition of a component or system as defined by the number, nature, and interconnections of its constituent parts.
2. State of the environment of a test object, which must be fulfilled as a precondition for executing the test cases.

**configuration item**

Software object or test environment that is under configuration management.

**configuration management**

Activities for managing the configurations.

**constructive quality assurance**

The use of methods, tools, and guidelines that contribute to making sure the following conditions are met:

- The product to be produced and/or the production process have certain attributes from before.
- Errors and mistakes are minimized or prevented.

**control flow**

An abstract representation of all possible sequences of events (paths) during execution of a component or system. Often represented in graphical form, see *control flow graph*.

**control flow anomaly**

Statically detectable anomaly in execution of a test object (for example, statements that aren't reachable).

**control-flow-based test**

Dynamic test, whose test cases are derived using the control flow of the test object and whose test coverage is determined against the control flow.

**control flow graph**

- A graphical representation of all possible sequences of events (paths) in the execution through a component or system.
- A formal definition: A directed graph  $G = (N, E, nstart, nfinal)$ .  $N$  is the finite set of nodes.  $E$  is the set of directed branches.  $nstart$  is the start node.  $nfinal$  is the end node. Control flow graphs are used to show the control flow of components.

**coverage**

Criterion for the intensity of a test (expressed as a percentage), differing according to test method. Coverage can usually be found by using tools.

**cyclomatic number**

Metric for complexity of a control flow graph. It shows the number of linearly independent paths through the control flow graph or a component represented by the graph.

**data quality**

The degree to which data in an IT system is complete, up-to-date, consistent, and (syntactically and semantically) correct.

**data flow analysis**

A form of static analysis that is based on the definition and usage of variables and shows wrong access sequences for the variable of the test object.

**data flow anomaly**

Unintended or unexpected sequence of operations on a variable.

**data flow coverage**

The percentage of definition-use pairs that have been executed by a test suite.

**data flow test techniques**

White box test design techniques in which test cases are designed using data flow analysis and where test completeness is assessed using the achieved data flow coverage.

**data security (security)**

Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization [ISO 25010].

From [ISO 9126]: The capability of the software product to protect programs and data from unauthorized access, whether this is done voluntarily or involuntarily.

**dead code**

See *unreachable code*.

**debugger**

A tool used by programmers to reproduce failures, investigate the state of programs, and find the corresponding defect. Debuggers enable programmers to execute programs step-by-step, to stop a program at any program statement, and to set and display program variables.

**debugging**

The process of finding, analyzing, and removing the causes of failures in software.

**decision coverage**

The percentage of decision outcomes that have been exercised by a test suite. 100% decision coverage implies both 100% branch coverage and 100% statement coverage.

See also *branch coverage*.

**decision table**

A table showing rules that consist of combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects). These tables can be used to design test cases.

**decision test(ing)**

Control-flow-based white box test design technique requiring that each decision outcome (TRUE and FALSE) is used at least once for the test

object. (An IF statement has two outcomes; a CASE or SWITCH statement has as many outcomes as there are given.)

**defect**

A flaw in a component or system that can cause it to fail to perform its required function, such as, for example, an incorrect statement or data definition. A defect, if encountered during execution, may cause a *failure* of the component or system.

**defect database**

1. List of all known defects in the system, a component, and their associated documents as well as their states.
2. A current and complete list with information about known failures.

**Defect Detection Percentage (DDP)**

The number of defects found at test time or in a test level divided by the number found altogether in that period plus the number found until a future defined time point in time (for example, six months after release).

**defect management**

The process of recognizing, investigating, taking action, and disposing of detected defects. It involves recording defects, classifying them, and identifying their impact.

**defect masking**

An occurrence in which one defect prevents the detection of another.

**development model**

*See software development model.*

**developer test**

A test that is under the (sole) responsibility of the developer of the test object (or the development team). Often seen as equal to component test.

**driver**

A program or tool that makes it possible to execute a test object, to feed it with test input data, and to receive test output data and reactions.

**dummy**

A special program, normally restricted in its functionality, to replace the real program during testing.

**dynamic analysis**

The process of evaluating the behavior (e.g., memory performance, CPU usage) of a system or component during execution.

**dynamic tests**

Tests that are executing code. Tests in a narrow sense, i.e., how the general public understands testing. The opposite are static tests. Static and dynamic tests taken together form the whole topic of testing.

**efficiency**

A set of software characteristics (for example, execution speed, response time) relating to performance of the software and use of resources (for example, memory) under stated conditions (normally increasing load).

**equivalence class**

See *equivalence partition*.

**equivalence partition**

A portion of an input or output domain for which the behavior of a component or system is assumed to be the same; judgment being based on the specification.

**equivalence class partitioning**

Partitioning input or output domains of a program into a limited set of classes, where the elements of a class show equivalent functional behavior.

From the ISTQB glossary: A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle, test cases are designed to cover each partition at least once.

**error**

A human action that produces an incorrect result. Also a general, informally used term for terms like *mistake*, *fault*, *defect*, *bug*, *failure*.

**error guessing**

A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made and to design tests specifically to expose them.

**exception handling**

Behavior of a component or system in response to wrong input, from either a human user or another component or system, or due to an internal failure.

**exhaustive testing**

A test approach in which the test suite comprises all combinations of input values and preconditions. Usually this is not practically achievable.

**exit criteria**

The set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task that have not been finished. Achieving a certain degree of test coverage for a white box test is an example of an exit criterion.

**expected result**

The predicted or expected output or behavior of a system or its components, as defined by the specification or another source (for every test case).

**exploratory testing**

An informal test design technique where the tester actively controls the design of the tests as those tests are performed. The tester uses the information gained while testing to design new and better tests.

**Extreme Programming**

A lightweight agile software engineering methodology used whereby a core practice is test-first programming.

**failure**

1. Deviation of the component or system from its expected delivery, service, or result. (For a test result, the observed result and the expected [specified or predicted] result do not match.)
2. Result of a fault that, during test execution, shows an externally observable wrong result.
3. Behavior of a test object that does not conform to a specified functionality that should be suitable for its use.

**failure class / failure classification / failure taxonomy**

Classification of the found failures by their severity from a user point of view (for example, the degree of impairment of product use).

**failure priority**

Determination of how urgent it is to correct the cause of a failure by taking into account failure severity, necessary correction work, and the effects on the whole development and test process.

**fault**

An alternative term for *defect*.

**fault masking**

A fault in the test object is compensated by one or more faults in other parts of the test object in such a way that it does not cause a failure. (Note: Such faults may then cause failures after other faults have been corrected.)

**fault revealing test case**

A test case that, when executed, leads to a different result than the specified or expected one.

**fault tolerance**

1. The capability of the software product or a component to maintain a specified level of performance in case of wrong inputs (see also *robustness*).
2. The capability of the software product or a component to maintain a specified level of performance in case of software faults (defects) or of infringement of its specified interface.

**field test(ing)**

Test of a preliminary version of a software product by (representatively) chosen customers with the goal of finding influences from incompletely known or specified production environments. Also a test to check market acceptance.

See also *beta testing*.

**finite state machine**

A computation model consisting of a limited number of states and state transitions, usually with corresponding actions. Also called *state machine*.

**functional requirement**

A requirement that specifies a function that a component or system must perform.

See also *functionality*.

**functional testing**

1. Checking functional requirements.
2. Dynamic test for which the test cases are developed based on an analysis of the functional specification of the test object. The completeness of this test (its coverage) is assessed using the functional specification.

**functionality**

The capability of the software product to provide functions that meet stated and implied needs when the software is used under specified conditions. Functionality describes WHAT the system must do. Implementation of functionality is the precondition for the system to be usable at all. Func-

tionality includes the following characteristics: suitability, correctness, interoperability, compliance, and security [ISO 9126].

**GUI**

Graphical user interface.

**incident database**

A collection of information about incidents, usually implemented as a database. An incident database shall make it possible to follow up incidents and extract data about them.

**informal review**

A review not based on a formal (documented) procedure.

**inspection**

A type of review that relies on visual examination of documents to detect defects—for example, violations of development standards and nonconformance to higher-level documentation. Inspection is the most formal review technique and therefore always based on a documented procedure.

**instruction**

See *statement*.

**instrumentation**

The insertion of additional logging or counting code into the source code of a test object (by a tool) in order to collect information about program behavior during execution (e.g., for measuring code coverage).

**integration**

The process of combining components or systems into larger assemblies.

**integration test(ing)**

Testing performed to expose defects in the interfaces and in the interactions between integrated components.

**level test plan**

A plan for a specified level of testing. It identifies the items being tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the associated risk(s). In the title of the plan, the word *level* is replaced by the organization's name for the particular level being documented by the plan (e.g., Component Test Plan, Component Integration Test Plan, System Test Plan, and Acceptance Test Plan).

**load test(ing)**

Measuring the behavior of a system as the load increases (e.g., increase in the number of parallel users and/or number of transactions) in order to determine what load can be handled by the component or system. (Load testing is a kind of performance testing.)

**logical test case**

A test case without concrete values for the input data and expected results. Usually, value ranges (equivalence classes) are given.

**maintainability**

The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment.

**maintenance / maintenance process**

Modification of a software product after delivery to correct defects, to improve performance or other attributes, or to adapt the product to a modified environment.

**management review**

1. A review evaluating project plans and development processes.
2. A systematic evaluation of software acquisition, supply, development, operation, or maintenance process performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches to achieve fitness for purpose.

**master test plan**

A detailed description of test objectives to be achieved and the means and schedule for achieving them, organized to coordinate testing activities for some test object or set of test objects. A master test plan may comprise all testing activities on the project; further detail of particular test activities could be defined in one or more test subprocess plans (for example, a system test plan or a performance test plan or level test plans).

**metric**

1. A value from measuring a certain program or component attribute. Finding metrics is a task for static analysis.
2. A measurement scale and the method used for measurement.

**milestone**

This marks a point in time in a project or process at which a defined result should be ready.

**mistake**

See *error*.

**mock-up, mock, mock object**

A program in the test environment that takes the place of a stub or dummy but contains more functionality. This makes it possible to trigger desired results or behavior.

**moderator**

The leader and main person responsible for an inspection or a review meeting.

**module testing**

Test of a single module of a modular software system.

See *component testing*.

**monitor**

A software tool or hardware device that runs concurrently with the component or system under test and supervises, records, analyzes, and/or verifies its behavior.

**multiple-condition testing**

Control-flow-based white box test design technique in which test cases are designed to execute all combinations of single-condition outcomes (true and false) within one decision statement.

**negative test(ing)**

1. Usually a functional test case with inputs that are not allowed (following the specification). The test object should react in a robust way, such as, for example, rejecting the values and executing appropriate exception handling.
2. Testing aimed at showing that a component or system does not work, such as, for example, a test with wrong input values. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique.

**nonfunctional requirement**

A requirement that does not directly relate to functionality but to how well or with which quality the system fulfills its function. Its implementation has a great influence on how satisfied the customer or user is with the sys-

tem. The attributes from [ISO 9126] are reliability, efficiency, usability, maintainability, and portability.

The attributes from [ISO 25010] are performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

**nonfunctional testing**

Testing the nonfunctional requirements.

**nonfunctional tests**

Tests for the nonfunctional requirements.

**patch**

1. A modification made directly to an object code without modifying the source code or reassembling or recompiling the source program.
2. A modification made to a source program as a last-minute fix or as an afterthought.
3. Any modification to a source or object program.
4. To perform a modification such as described in the preceding three definitions.
5. Unplanned release of a software product with corrected files in order to, possibly in a preliminary way, correct special (often blocking) faults.

**path**

1. A path in the program code: A sequence of events (e.g., executable statements) of a component or system from an entry point to an exit point.
2. A path in the control flow graph: An alternating sequence of nodes and branches in a control flow graph. A complete path starts with the node “nstart” and ends with the node “nfinal” of the control flow graph. Executable and not executable paths can be distinguished.

**path testing, path coverage**

Control-flow-based white-box test design technique that requires executing all different complete paths of a test object. In practice, this is usually not feasible due to the large number of paths.

**performance**

The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate. In [ISO 25010] called performance efficiency.

**performance testing**

The process of testing to determine the performance of a software product for certain use cases, usually dependent on increasing load.

**Point of Control (PoC)**

Interface used to send test data to the test object.

**Point of Observation (PoO)**

Interface used to observe and log the reactions and outputs of the test object.

**postconditions**

Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.

**precondition**

Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.

**preventive software quality assurance**

Use of methods, tools, and procedures that contribute to designing quality into the product. As a result of their application, the product should then have certain desired characteristics, and faults are prevented or their effects minimized.

Note: Preventive (constructive) software quality assurance is often used in early stages of software development. Many defects can be avoided when the software is developed in a thorough and systematic manner.

**problem**

See *defect*.

**problem database**

1. A list of known failures or defects/faults in a system or component and their state of repair.
2. A database that contains current and complete information about all identified defects.

**process model**

See *software development model*.

**production environment**

The hardware and software products, as well as other software with its data content (including operating systems, database management systems and other applications), that are in use at a certain user site.

This environment is the place where the test object will be operated or used.

**quality**

1. The totality of characteristics and their values relating to a product or service. They relate to the product's ability to fulfill specified or implied needs.
2. The degree to which a component, system, or process meets user/customer needs and expectations.
3. The degree to which a set of inherent characteristics fulfills requirements.

**quality assurance**

All activities within quality management focused on providing confidence that quality requirements are fulfilled.

**quality attribute**

1. A characteristic of a software product used to judge or describe its quality. A software quality attribute can also be refined through several steps into partial attributes.
2. Ability or characteristic which influences the quality of a unit.

**quality characteristic**

See *quality attribute*.

**random testing**

A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile in the production environment. Note: This technique can, among others, be used for testing nonfunctional attributes such as reliability and performance.

**regression testing**

Testing a previously tested program or a partial functionality following modification to show that defects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made. It is performed when the software or its environment is changed.

**release**

A particular version of a configuration item that is made available for a specific purpose, such as, for example, a test release or a production release.

**reliability**

A set of characteristics relating to the ability of the software product to perform its required functions under stated conditions for a specified period of time or for a specified number of operations.

**requirement**

A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

**requirements-based testing**

An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, such as, for example, tests that exercise specific functions or probe nonfunctional attributes such as reliability or usability.

**requirements definition**

1. Written documentation of the requirements for a product or partial product to be developed. Typically, the specification contains functional requirements, performance requirements, interface descriptions, design requirements, and development standards.
2. Phase in the general V-model in which the requirements for the system to be developed are collected, specified, and agreed upon.

**retesting**

Testing that executes test cases that failed the last time they were run in order to verify the success of correcting faults.

**review**

1. Measuring, analyzing, checking of one or several characteristics of a unit (the review object), and comparing with defined requirements in order to decide if conformance for every characteristic has been achieved.
2. Abstract term for all analytical quality assurance measures independent of method and object.
3. An evaluation of a product or project status to ascertain discrepancies of the planned work results from planned results and to recommend improvements. Reviews include management review, informal review, technical review, inspection, and walkthrough.

**reviewable (testable)**

A work product or document is reviewable or testable if the work is complete enough to enable a review or test of it.

**risk**

A factor that could result in future negative consequences; usually expressed as impact and likelihood.

**risk-based testing**

An approach to testing to reduce the level of product risks and inform stakeholders of their status, starting in the initial stages of a project. It involves the identification of product risks and the use of risk levels to guide the test process.

**robustness**

The degree to which a component or system can function correctly in the presence of invalid inputs or stressful or extreme environmental conditions.

**robustness test(ing)**

See *negative testing*.

**role**

Description of specific skills, qualifications and work profiles in software development. These should be filled by the persons (responsible for these roles) in the project.

**safety-critical system**

A system whose failure or malfunction may result in death or serious injury to people, loss or severe damage to equipment, or environmental harm.

**security test**

Testing to determine the access or data security of the software product. Also testing for security deficiencies.

**severity**

The degree of impact that a defect has on the development or operation of a component or system.

**simulator**

1. A tool with which the real or production environment is modeled.
2. A system that displays chosen patterns of behavior of a physical or abstract system.

**smoke test**

1. Usually an automated test (a subset of all defined or planned test cases) that covers the main functionality of a component or system in order to ascertain that the most crucial functions of a program work but not bothering with finer details.
2. A smoke test is often implemented without comparing the actual and the expected output. Instead, a smoke test tries to produce openly visible wrong results or crashes of the test object. It is mainly used to test robustness.

**software development model/software development process**

Model or process that describes a defined organizational framework of software development. It defines which activities shall be executed by which roles in which order, which results will be produced, and how the results are checked by quality assurance.

**software item**

Identifiable (partial) result of the software development process (for example, a source code file, document, etc.).

**software quality**

The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

**specification**

A document that specifies, ideally in a complete, precise, concrete and verifiable form, the requirements or other characteristics of a component or system. It serves the developers as a basis for programming, and it serves the testers as a basis for developing test cases with black box test design methods. (Often, a specification includes procedures for determining whether these requirements have been satisfied.)

**special software**

Software developed for one or a group of customers. The opposite is standard or commercial off-the-shelf (COTS) software. The British term is *bespoke software*.

**state diagram**

A diagram or model that describes the states that a component or system can assume and that shows the events or circumstances that cause and/or result from a change from one state to another.

**state transition testing**

A black box test design technique in which test cases are designed to execute valid and invalid state transitions of the test object from the different states. The completeness (coverage) of such a test is judged by looking at the states and state transitions.

**statement**

A syntactically defined entity in a programming language. It is typically the smallest indivisible unit of execution. Also referred to as an *instruction*.

**statement test(ing)**

Control-flow-based test design technique that at the least requires that every executable statement of the program has been executed once.

**statement coverage**

The percentage of executable statements that have been exercised by a test suite.

**static analysis**

Analysis of a document (e.g., requirements or code) carried out without executing it.

**static analyzer**

A tool that carries out static analysis.

**static testing**

Testing of a component or system at the specification or implementation level without execution of any software (e.g., using reviews or static analysis).

**stress testing**

Test of system behavior with overload. For example, running it with too high data volumes, too many parallel users, or wrong usage.

See also *robustness*.

**structural test(ing), structure-based test(ing)**

White box test design technique in which the test cases are designed using the internal structure of the test object. Completeness of such a test is judged using coverage of structural elements (for example, branches, paths, data). General term for *control-* or *data-flow-based test*.

**stub**

A skeletal or special-purpose implementation of a software component, needed in component or integration testing and used to replace or simulate not-yet-developed components during test execution.

**syntax testing**

A test design technique in which test cases are designed based on a formal definition of the input syntax.

**system integration testing**

Testing the integration of systems (and packages); testing interfaces to external organizations (e.g., Electronic Data Interchange, Internet).

**system testing**

The process of testing an integrated system to ensure that it meets specified requirements.

**technical review**

A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. A technical review is also known as a peer review.

**test**

A set of one or more test cases

**test automation**

1. The use of software tools to design or program test cases with the goal to be able to execute them repeatedly using the computer.
2. To support any test activities by using software tools.

**test basis**

All documents from which the requirements of a component or system can be inferred. The documentation on which the design and choice of the test cases is based.

**test bed**

An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. Also called *test environment*. Also used as an alternative term for *test harness*.

**test case**

A set of input values, execution preconditions, expected results, and execution postconditions developed for a particular objective or test con-

dition, such as to exercise a particular program path or to verify compliance with a specific requirement.

**test case explosion**

Expression for the exponentially increasing work for an exhaustive test with increasing numbers of parameters.

**test case specification**

A document specifying a set of test cases.

**test condition**

An item or event of a component or system that can be verified by one or more test cases, such as, for example, a function, transaction, feature, quality attribute, or structural element.

**test coverage**

See *coverage*.

**test cycle**

1. Execution of the fundamental test process against a single identifiable release of the test object. Its end result are orders for defect corrections or changes.
2. Execution of a series of test cases.

**test data**

1. Input or state values for a test object and the expected results after execution of the test case.
2. Data that exists (for example, in a database) before a test is executed and that affects or is affected by the component or system under test.

**test design technique**

A planned procedure (based on a set of rules) used to derive and/or select test cases. There are specification-based, structure-based, and experience-based test design techniques. Also called *test technique*.

**test driver**

See *driver*.

**test effort**

The resources (to be estimated or analyzed) for the test process.

**test environment**

See *test bed*.

**test evaluation**

Analysis of the test protocol or test log in order to determine if failures have occurred. If necessary, these are assigned a classification.

**test execution**

The process of running test cases or test scenarios (an activity in the test process) that produce actual result(s).

**test harness (test bed)**

A test environment comprising all stubs and drivers needed to execute test cases. Even logging and evaluation tasks may be integrated into a test harness.

**test infrastructure**

The artifacts needed to perform testing, consisting usually of test environments, test tools, office environment for the testers and its equipment, and other tools (like mail, Internet, text editors, etc.).

**testing**

The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation, and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

**test interface**

See *Point of Control (PoC)* and *Point of Observation (PoO)*.

**test level**

A group of test activities that are executed and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test, and acceptance test (from the generic V-model).

**test log**

The written result of a test run or a test sequence (in the case of automated tests often produced by test tools). From the log it must be possible to see which parts were tested when, by whom, how intensively, and with what result.

**test logging**

The process of recording information about tests executed into a test log.

**test management**

1. The planning, estimating, monitoring, control, and evaluation of test activities, typically carried out by a test manager.
2. Group of persons responsible for a test.

**test method**

See *test design technique*.

**test metric**

A measurable attribute of a test case, test run, or test cycle, including measurement instructions.

**test object**

The component, integrated partial system, or system (in a certain version) that is to be tested.

**test objective**

A reason or purpose for designing and executing a test. Examples are as follows:

1. General objective: Finding defects.
2. Finding special defects using suitable test cases.
3. Showing that certain requirements are fulfilled in or by the test object as a special objective for one or more test cases.

**test oracle**

An information source to determine expected results of a test case (usually the requirements definition or specifications).

A test oracle may also be an existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but it should not be the code. This is because then the code is used as a basis for testing, thus it is tested against itself.

**test phase**

A set of related activities (in the test process) that are intended for the design of an intermediate work product (for example, design of a test specification). This term differs from *test level*!

**test plan**

A document describing the scope, approach, resources, and schedule of intended test activities (from [IEEE 829-1998]).

It identifies, among other test items, the features to be tested, the testing tasks, who will do each task, the degree of tester independence, the test environment, the test design techniques, and the techniques for measuring

results with a rationale for their choice. Additionally, risks requiring contingency planning are described. Thus, a test plan is a record of the test planning process.

The document can be divided into a master test plan or a level test plan.

**test planning**

The activity of establishing or updating a test plan.

**test procedure / test script / test schedule**

1. Detailed instructions about how to prepare, execute, and evaluate the results of a certain test case.
2. A document specifying a sequence of actions for the execution of a test.

**test process**

The fundamental test process comprises all activities necessary for the test in a project, such as test planning and control, test analysis and design, test implementation and execution, evaluation of exit criteria and reporting, and test closure activities.

**test report**

An alternative term for *test summary report*.

**test result**

1. All documents that are written during a test cycle (mainly the test log and its evaluation).
2. Release or stopping of a test object (depending on the number and severity of failures discovered).

**test robot**

A tool to execute tests that uses open or accessible interfaces of the test objects (for example, the GUI) to feed in input values and read their reactions.

**test run**

Execution of one or several test cases on a specific version of the test object.

**test scenario**

A set of test sequences.

**test schedule**

1. A list of activities, tasks, or events of the test process identifying their intended start and finish dates and/or times and interdependencies (among others, the assignment of test cases to testers).
2. List of all test cases, usually grouped by topic or test objective.

**test script**

Instructions for the automatic execution of a test case or a test sequence (or higher-level control of further test tools) in a suitable programming language.

**test specification**

1. A document that consists of a test design specification, test case specification and/or test procedure specification.
2. The activity of specifying a test, typically part of “test analysis and design” in the test life cycle.

**test strategy**

1. Distribution of the test effort over the parts to be tested or the quality characteristics of the test object that should be fulfilled. Selection and definition of the order (or the interaction) of test methods and the order of their application on the different test objects. Definition of the test coverage to be achieved by each test method.
2. Abstract description of the test levels and their corresponding start and exit criteria. Usually, a test strategy can be used for more than one project.

**test suite / test sequence**

A set of several test cases in which the postcondition of one test is often used as the precondition for the next one.

**test summary report**

A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria. Also called *test report*.

**test technique**

1. See *test design technique*.
2. A combination of activities to systematically design a test work product. In addition to designing test cases, test techniques are available for activities such as test estimation, defect management, product risk analysis, test execution, and reviews.

**testability**

1. Amount of effort and speed with which the functionality and other characteristics of the system (even after each maintenance) can be tested.
2. Ability of the tested system to be tested. (Aspects are openness of the interface, documentation quality, ability to partition the system into smaller units, and ability to model the production environment in the test environment.)

**tester**

1. An expert in test execution and reporting defects with knowledge about the application domain of the respective test object.
2. A general term for all people working in testing.

**test-first programming**

Software development process where test cases defining small controllable implementation steps are developed before the code is developed. Also called test-first design, test-first development, test-driven design, or test-driven development.

**testing**

The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

**testware**

All documents and possibly tools that are produced or used during the test process and are required to plan, design, and execute tests. Such documents may include scripts, inputs, expected results, setup and clean-up procedures, files, databases, environment, and any additional software or utilities used in testing. Everything should be usable during maintenance and therefore must be transferable and possible to update.

**traceability**

The ability to identify related items in documentation and software, especially requirements with associated tests.

**tuning**

Changing programs or program parameters and/or expanding hardware to optimize the time behavior of a hardware/software system.

**unit testing**

See *component testing*.

**unit test**

See *component test*.

**unnecessary test**

A test that is redundant with another already present test and thus does not lead to new results.

**unreachable code**

Code that cannot be reached and therefore is impossible to execute.

**use case**

A sequence of transactions in a dialogue between an actor and a component or system with a tangible result. An actor can be a user or anything that can exchange information with the system.

**use case testing**

A black box test design technique in which test cases are designed to execute scenarios of use cases.

**user acceptance test**

An *acceptance test* on behalf of or executed by users.

**validation**

Testing if a development result fulfills the individual requirements for a specific use.

**verification**

1. Checking if the outputs from a development phase meet the requirements of the phase inputs.
2. Mathematical proof of correctness of a (partial) program.

**version**

Development state of a software object at a certain point of time. Usually given by a number.

See also *configuration*.

**V-model (generic)**

A framework to describe the software development life cycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development life cycle and how intermediate products can be verified and validated. Many different variants of the V-model exist nowadays.

**volume testing**

Testing in which large amounts of data are manipulated or the system is subjected to large volumes of data.

See also *stress testing* and *load testing*.

**walkthrough**

A manual, usually informal review method to find faults, defects, unclear information, and problems in written documents. A walkthrough is done using a step-by-step presentation by the author. Additionally, it serves to gather information and to establish a common understanding of its content. Note: For ISTQB purposes, a walkthrough is a *formal* review as opposed to the *informal review*.

**white box test design technique**

Any technique used to derive and/or select test cases based on an analysis of the internal structure of the test object (see also *structural test*).

---

# Literature

- [Adrion 82] Adrion, W.; Branstad, M.; Cherniabsky, J.: “Validation, Verification and Testing of Computer Software,” *Computing Surveys*, Vol. 14, No 2, June 1982, pp. 159–192.
- [Bach 04] Bach, J.: “Exploratory Testing,” in [van Veenendaal 04], pp. 209–222.
- [Bashir 99] Bashir, I.; Paul, R.A.: “Object-oriented integration testing,” *Automated Software Engineering*, Vol. 8, 1999, pp. 187–202.
- [Bath 08] Bath, G.; McKay, J.: *The Software Test Engineer’s Handbook. A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates*, Rocky Nook, 2008.
- [Bath 13] Bath, G.; Veenendaal, E.V.: *Improving the Test Process. Implementing Improvement and Change - A Study Guide for the ISTQB Expert Level Module*, Rocky Nook, 2013.
- [Bath 14] Bath, G.; McKay, J.: *The Software Test Engineer’s Handbook. A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates*, 2nd edition, Rocky Nook, 2014.
- [Beck 00] Beck, K.: *Extreme Programming*, Addison-Wesley, 2000.
- [Beedle 01] Beedle, M.; Schwaber, K.: *Agile Software Development with SCRUM*, Prentice Hall, 2001.
- [Beizer 90] Beizer, B.: *Software Testing Techniques*, Van Nostrand Reinhold, 1990.
- [Beizer 95] Beizer, B.: *Black-Box Testing*, John Wiley & Sons, 1995.
- [Binder 99] Binder, R.V.: *Testing Object-Oriented Systems*, Addison-Wesley, 1999.
- [Black 02] Black, R.: *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, 2nd ed., John Wiley & Sons, 2002.
- [Black 03] Black, R.: *Critical Testing Processes*, Addison-Wesley, 2003.
- [Black 08] Black, R.: *Advanced Software Testing—Vol. 1, Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*, Rocky Nook, 2008.
- [Black 09] Black, R.: *Advanced Software Testing—Vol. 2, Guide to the ISTQB Advanced Certification as an Advanced Test Manager*, Rocky Nook, 2009.
- [Black 11] Black, R.: *Advanced Software Testing—Vol. 3, Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*, Rocky Nook, 2011.
- [Bleek 08] Bleek, W.-G.; Henning, W.: *Agile Softwareentwicklung*, dpunkt.verlag, 2008.
- [Boehm 73] Boehm, B. W.: “Software and Its Impact: A Quantitative Assessment,” *Data-mation*, Vol 19, No. 5, 1973, pp. 48–59.
- [Boehm 79] Boehm, B. W.: “Guidelines for Verifying and Validation Software Requirements and Design Specifications,” *Proceedings of Euro IFIP 1979*, pp. 711–719.

- [Boehm 81] Boehm, B. W.: *Software Engineering Economics*, Prentice Hall, 1981.
- [Boehm 86] Boehm, B. W.: "A Spiral Model of Software Development and Enhancement," ACM SIGSOFT, August 1986, pp. 14–24.
- [Bourne 97] Bourne, K. C.: *Testing Client/Server Systems*, McGraw-Hill, 1997.
- [Bush 90] Bush, M.: "Software Quality: The use of formal inspections at the Jet Propulsion Laboratory," Proceedings of the 12th ICSE, IEEE 1990, pp. 196–199.
- [Chow 78] Chow, T.: "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, Vol. 4, No 3, May 1978, pp. 178–187.
- [Clarke et al. 85] Clarke, L.A.; Podgurski, A.; Richardson, D.J.; Zeil, S.J.: "A Comparison of Data Flow Path Selection Criteria," Proceedings of the 8th International Conference on Software Engineering, August 1985, pp. 244–251.
- [CMMI 02] Capability Maturity Model<sup>®</sup> Integration, Version 1.1, CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMISE/SW/IPPD/SS, V1.1), Staged Representation, CMU/SEI-2002-TR-012, 2002.
- [DeMarco 93] DeMarco, T.: "Why Does Software Cost So Much?," *IEEE Software*, March 1993, pp. 89–90.
- [Fagan 76] Fagan, M. E.: "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182–211.
- [Fenton 91] Fenton, N. E.: *Software Metrics*, Chapman&Hall, 1991.
- [Fewster 99] Fewster, M.; Graham, D.: *Software Test Automation, Effective use of test execution tools*, Addison-Wesley, 1999.
- [Freedman 90] Freedman, D. P.; Weinberg, G. M.: *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, 3rd ed., Dorset House, 1990.
- [Gerrard 02] Gerrard, P.; Thompson, N.: *Risk-Based E-Business Testing*, Artech House, 2002.
- [Gilb 96] Gilb, T.; Graham, D.: *Software Inspections*, Addison-Wesley, 1996.
- [Gilb 05] Gilb, T.: *Competitive Engineering: A Handbook for Systems & Software Engineering Management Using Planguage*, Butterworth-Heinemann, Elsevier, 2005.
- [Hetzel 88] Hetzel, W. C.: *The Complete Guide to Software Testing*, 2nd ed., John Wiley & Sons, 1988.
- [Howden 75] Howden, W.E.: "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers*, Vol. 24, No. 5, May 1975, pp. 554–560.
- [Jacobson 99] Jacobson, I.; Booch, G., Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Koomen 99] Koomen, T.; Pol, M.: *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999.
- [Kung 95] Kung, D.; Gao, J.; Hsia, P.: "On Regression Testing of Object-Oriented Programs," *Journal of Systems and Software*, Vol. 32, No. 1, Jan 1995, pp. 21–40.
- [Link 03] Link, J.: *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufmann, 2003.
- [Martin 91] Martin, J.: *Rapid Application Development*, Macmillan, 1991.
- [McCabe 76] McCabe, T. J.: "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 1976, pp. 308–320.
- [Musa 87] Musa, J.: *Software Reliability Engineering*, McGraw-Hill, 1998.

- [Myers 79] Myers, G.: *The Art of Software Testing*, John Wiley & Sons, 1979.
- [Pol 98] Pol, M.; van Veenendaal, E.: *Structured Testing of Information Systems – an Introduction to Tmap*, Kluwer, 1998.
- [Pol 02] Pol, M.; Teunissen, R.; van Veenendaal, E.: *Software Testing, A Guide to the TMap Approach*, Addison-Wesley, 2002.
- [Rothermel 94] Rothermel, G; Harrold, M.-J.: “Selection Regression Test for Object-Oriented Software,” Proceedings of the International Conference on Software Maintenance, 1994, pp. 14–25.
- [Royce 70] Royce, W. W.: “Managing the development of large software systems,” IEEE WESCON, Aug. 1970, pp. 1–9 (reprinted in Proceedings of the 9th ICSE, 1987, Monterey, CA., pp. 328–338).
- [Spillner 08] Spillner, A.; Rossner, T.; Winter, M.; Linz, T.: *Software Testing Practice: Test Management. A Study Guide for the Certified Tester Exam ISTQB Advanced Level*. Rocky Nook, Santa Barbara, 2008.
- [Spillner 00] Spillner, A.: “From V-model to W-model – Establishing the Whole Test Process”, Proceedings Conquest 2000 – Workshop on “Testing Nonfunctional Software Requirements,” Sept. 2000, Nuremberg, pp. 221-231.
- [Stapleton 02] Stapleton, J. (ed.): *DSDM: Business Focused Development* (Agile Software Development Series), Addison-Wesley, 2002.
- [van Veenendaal 04] van Veenendaal, E. (ed.): *The Testing Practitioner*, UTN Publishers, 2004.
- [Vigenschow 2010] Vigenschow, U.: *Testen von Software und Embedded Systems*, dpunkt.verlag, Heidelberg, 2nd ed., 2010.
- [Winter 98] Winter, M.: Managing Object-Oriented Integration and Regression Testing, Proceeding of the 6th euroSTAR 98, Munich, 1998, pp. 189–200.

## Further Recommended Literature

- Buwalda, H.; Jansson, D.; Pinkster, I.: *Integrated Test Design and Automation, Using the TestFrame Methods*, Addison-Wesley, 2002.
- Dustin, E.; Rashka, J.; Paul, J.: *Automated Software Testing, Introduction, Management and Performance*, Addison-Wesley, 1999.
- Jorgensen, Paul C.: *Software Testing – A Craftman’s Approach*, 2nd ed., CRC Press, 2002.
- Kaner C.; Falk, J.; Nguyen, H. Q.: *Testing Computer Software*, 2nd ed., John Wiley & Sons, 1999.
- Kit, E.: *Testing in the Real World*, Addison-Wesley, 1995.
- Ould, M. A.; Unwin, C.: (ed.): *Testing in Software Development*, Cambridge University Press, 1986.
- Perry, W. E.: *Effective Methods for Software Testing*, John Wiley & Sons, 2000.
- Roper, M.: *Software Testing*, McGraw-Hill, 1994.
- Royer, T. C.: *Software Testing Management*, Prentice Hall, 1993.
- Whittaker, J.: *How to Break Software*, Addison-Wesley, 2003.

## Standards

- [BS 7925-1] British Standard BS 7925-1, Software Testing, Part 1: Vocabulary, 1998.
- [BS 7925-2] British Standard BS 7925-2, Software Testing, Part 2: Software Component Testing, 1998. This standard was the basis for the British Computer Society ISEB certification and the earlier version of the ISTQB certification. It will be revised and, over time, superseded by ISO/IEC 29119.
- [EN 50128] EN 50128:2001, Railway applications – Communication, signaling and processing systems – Software for railway control and protection systems, European Committee for Electrotechnical Standardization.
- [IEEE 610.12] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology. Superseded by IEEE Std. 24765-2010 – IEEE Systems and software engineering – Vocabulary.
- [IEEE 730-2002] IEEE Std 730-2002, IEEE Standard for Software Quality Assurance Plans.
- [IEEE 730-2013] IEEE Std 730-2013, IEEE Standard for Software Quality Assurance Processes.
- [IEEE 828] IEEE Std 828-2012, IEEE Standard for Configuration Management in Systems and Software Engineering.
- [IEEE 829] IEEE Std 829-1998, IEEE Standard for Software Test Documentation (under revision, new edition probably in 2006).
- [IEEE 829-2008] IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation (revision of IEEE Std 829-1998).
- [IEEE 830] IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications.
- [IEEE 982] IEEE Std 982.2-2003, IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.
- [IEEE 1008] IEEE Std 1008-1987, IEEE Standard for Software Unit Testing.
- [IEEE 1012] IEEE Std 1012-2012, IEEE Standard for System and Software Verification and Validation.
- [IEEE 1028] IEEE Std 1028-2008, IEEE Standard for Software Reviews and Audits.
- [IEEE 1044] IEEE Std 1044-2009, IEEE Standard Classification for Software Anomalies.
- [IEEE 1219] IEEE Std 1219-1998, IEEE Standard for Software Maintenance.
- [IEEE/IEC 12207] IEEE/EIA Std 12207-2008: Information Technology – Software life cycle processes.
- [ISO 9000] ISO 9000:2005, Quality management systems – Fundamentals and vocabulary
- [ISO 9001] ISO 9001:2008, Quality management systems – Requirements.
- [ISO 90003] ISO/IEC 90003:2004, Software Engineering – Guidelines for the application of ISO 9001:2000 to computer software.
- [ISO 9126] ISO/IEC 9126-1:2001, Software Engineering – Product quality – Part 1: Quality model, Quality characteristics and sub characteristics.
- [ISO 9241] ISO 9241-1:2002-02 (D), Ergonomic requirements for office work with visual display terminals (VDTs) – Part 1: General introduction. (ISO 9241-1:1997), contains revision AMD 1:2001.

- [ISO 14598] ISO/IEC 14598-1:1996, Information Technology – Software Product Evaluation – Part 1: General Overview. This standard has been revised by: ISO/IEC 25040:2011
- [ISO 25010] ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.
- [ISO 25012]. ISO/IEC 25012:2008, Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Data quality model
- [ISO 29119] This is a new series of standards on software testing. This series is basis of the syllabus valid from 2015.
- ISO/IEC/IEEE 29119-1:2013 Software and systems engineering – Software testing – Part 1: Concepts and definitions
- ISO/IEC/IEEE 29119-2:2013 Software and systems engineering – Software testing – Part 2: Test processes
- ISO/IEC/IEEE 29119-3:2013 Software and systems engineering – Software testing – Part 3: Test documentation
- ISO/IEC/IEEE 29119-4 (Draft International Standard in Feb 2014) Standard Systems and software engineering—Software testing—Part 4: Test techniques
- [RTCA-DO 178] RTCA-DO Std 178B, Radio Technical Commission for Aeronautics, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc., 1992. DO178C Software Considerations in Airborne Systems and Equipment Certification (new Version “C” of the standard from 2010).

## WWW Pages

- [URL: ACM Ethics] <http://www.acm.org/about/code-of-ethics> The Association for Computing Machinery (ACM) Code of Ethics and Professional Conduct.
- [URL: BCS] <http://www.bcs.org/> The BCS, British Computer Society.
- [URL: BCS SIGIST] <http://www.testingstandards.co.uk/> Website of the BCS SIGIST Standards Working Party.
- [URL: FDA] <http://www.fda.gov/> US Food and Drug Administration.
- [URL: FMEA] <http://de.wikipedia.org/wiki/FMEA> Failure Mode and Effects Analysis.
- [URL: Graham] <http://www.dorothygraham.co.uk> Home page of Dorothy Graham, A.B., M.Sc.
- [URL: GTB] <http://www.german-testing-board.info/englisch.html/> German Testing Board.
- [URL: HTML] <http://www.w3.org/html/wg/> Hypertext Markup Language Homepage definition by the World Wide Web Consortium (W3C).
- [URL: IEEE] <http://standards.ieee.org> Information about IEEE standards.
- [URL: IEEE Ethics] <http://www.ieee.org/about/corporate/governance/p7-8.html> The IEEE Code of Ethics.
- [URL: ISEB] <http://www.iseb.org.uk> Information Systems Examination Board (ISEB).
- [URL: ISO] <http://www.iso.org/> The International Organization for Standardization (ISO) website.

- [URL: ISTQB] <http://www.istqb.org> International Software Testing Qualifications Board.
- [URL: ISTQB Members] <http://www.istqb.org/> ISTQB Worldwide lists all current international ISTQB members.
- [URL: NIST Report] <http://www.nist.gov/director/planning/upload/report02-3.pdf> The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards & Technology, USA, May 2002.
- [URL: pairwise] <http://www.pairwise.org/> This website contains information about the pairwise combination testing technique and links to opensource, freeware, and commercial tools to support it.
- [URL: Parnas] [http://en.wikipedia.org/wiki/David\\_Parnas](http://en.wikipedia.org/wiki/David_Parnas) Information about Dr. David Lorge Parnas.
- [URL: Pol] <http://www.polteq.com/en/over-polteq/martin-pol> Information by and about Martin Pol.
- [URL: RBS] <http://www.rbc-us.com/software-testing-resources/articles> RBCS Library (papers by Rex Black and others).
- [URL: rockynook] Rocky Nook, the publisher of this book. Rocky Nook maintains a website with updates for this book at [URL: softwaretest-knowledge].
- [URL: Schaefer] <http://www.softwaretesting.no> Home page of Hans Schaefer.
- [URL: SEPT] <http://www.12207.com/index.html> Software Engineering Process Technology Company (SEPT), Supplying Software Engineering Standards Information to the World.
- [URL: softwaretest-knowledge] <http://www.softwaretest-knowledge.de> Website with information about this book, its English and German corollary, and other books for the Certified Tester to further their education.
- [URL: SWEBOK] <http://www.computer.org/portal/web/swebok> Guide to the Software Engineering Body of Knowledge, SWEBOK V3.0, 2014.
- [URL: TestBench] <http://www.imbus.de/english/imbus-testbench/> Website for imbus TestBench.
- [URL: Tool-List] <http://www.imbus.de/english/test-tool-list/> Overview of testing tools; See also <http://opensourcetesting.org>, Open Source Tools for Software Testing Professionals.
- [URL: UML] <http://www.uml.org/> Home page of Object Management Group (OMG).
- [URL: V-model XT] <http://fouever.sourceforge.net/>. The flexible V-model can be downloaded here.
- [URL: XML] <http://www.w3.org/XML/> Extensible Markup Language home page of the World Wide Web Consortium (W3C)
- [URL: xunit] <http://www.junit.org> Component testing framework for Java. See also <http://opensourcetesting.org/>, Open Source Tools for Software Testing Professionals.

# Index

## A

- acceptance 62
- acceptance test 41, 61, 62, 77
- acceptance test environment 62
- accreditation 2
- actual result 7, 36
- ad-hoc-integration 57
- adaptability 13
- adequacy 11
- alpha test 64
- analysator 95–97
- analysis 79, 95, 96
  - static A. 13, 95, 97
- analysis tool 97
- analyzability 13
- anomaly 98, 99
- atomic partial condition 152
- audit 202
- author 84, 86, 102

## B

- backbone integration 57
- BCS (British Computer Society) 2
- bespoke software 252
- beta test 64
- black box technique 23, 108, 110
- blocked test 190
- bottom-up integration 57
- boundary value analysis 121
- branch test 146
- buddy testing 45
- business process analysis 71
- business process-based test 71

## C

- capture-replay tool 212
- CASE tool 206
- CAST tool 206
- cause-effect graph 136
- cause-effect-graph analysis 136
- certificate 3
- certification program for software testers 2
- certified tester 3, 173, 245
- certified tester examination 245
- change 199, 200
- change control board 199
- change related test 77
- change request 81, 199, 201, 208, 254
- changeability 13
- checklist oriented testing 187
- checklists 83, 86
- checkpoint 213
- class 40
- class test 42
- client/server system 216
- code 208
- code based test 145
- code change 208
- code coverage 166, 179, 262
- code of ethics 35
- command-driven test 214
- comparator 215
- compiler 97
- complete test 13
- compliance 91, 97
- component specification 40
- component test 15, 40 ff, 77, 170
- components 178

concrete test case 23, 25, 130  
condition 146, 149, 151, 152, 153  
condition coverage  
    multiple 152  
    simple 146  
condition test 255, 264  
configuration 50, 60, 200  
configuration audit 201  
configuration management 200, 201, 203  
configuration object 201  
conformity 13, 92  
control flow 14  
control flow analysis 99  
control flow graph 14, 146  
control flow-based test 145  
conventions 96  
correctness 12  
cost aspects 180  
cost based metrics 190  
coverage 119, 215  
coverage analysis 215  
coverage analyzer 215, 218  
customer 40, 58, 61, 65, 200

## D

damage 187  
    probability 187  
data flow analysis 98  
data flow anomaly 97  
data flow based test 16  
data quality 59, 188, 217  
data security 64, 73  
data-driven test 214  
dead program statements 28  
debugger 212  
debugging 8  
decision table 136, 138  
decision test 148, 149, 257  
defect 6, 7, 46, 52, 65, 67, 208  
defect cause 7  
defect class 29, 190  
defect classification 195  
defect correction 190

defect correction cost 180  
defect cost 180  
    direct 180  
    indirect 180  
defect database 191  
defect finding rate 30  
defect management 192, 198, 203  
defect masking (fault masking) 7  
defect priority 196  
defect status (or state) 195, 197, 201  
defect status models 207, 208  
defined condition test 146  
degree of coverage 109, 120, 149  
delivery (release) 191  
developer test 45  
development model 17, 39, 69  
development process 17, 39, 79, 86, 88, 182  
direct failure cost 180  
dummy 105  
dynamic test 105  
    tool 211

## E

ease of learning 12  
economic aspects 180  
efficiency 11, 13, 47  
employee qualification 87, 172–174  
environment 43, 53, 59, 175, 179, 182, 186, 188, 191,  
    237, 241, 266, 272, 273  
equivalence class 114  
equivalence class partitioning 110  
error handling 72  
error message 64, 193, 208  
error tolerance 12  
error, mistake 6, 15, 26  
evaluation 221  
exception handling 46  
exit criteria 19, 28, 37, 102, 105  
expected behavior 7, 22, 123, 135  
experience based test design 105, 161  
exploratory testing 23, 161  
extreme programming 17, 69, 94

**F**

failure 6, 7, 15, 16, 26  
failure analysis 116, 180  
failure based metrics 190  
fault (or defect) 7  
fault revealing test case 261  
field test 64  
follow-up 85, 89  
functional system design 40  
functional test 46, 70  
functionality 11  
fundamental test process 17

**G**

Generic V-model 210  
German Testing Board 281  
GUI (graphical user interface) 215

**I**

implementation 40, 46  
incremental development 68  
indirect defect cost 180  
informal review 92  
inspection 80, 90, 92  
inspector 87, 90  
instrumentation 160  
integration strategy 55  
integration test 41, 53, 57, 171  
interoperability 12  
intuitive test case design 109  
ISEB 2  
ISTQB 2

**K**

keyword-based testing 214  
keyword-driven test 214

**L**

level test plan 177  
load test 10, 72, 216  
logical test case 23

**M**

maintenance 31  
management review 89  
mass test 95  
master test plan 177  
maturity 12, 182, 183  
maturity level 218  
measures 28  
metrics 190  
    cost based 190  
    failure based 190  
    test case based 190  
    test object based 190  
milestone 17  
mock 105  
model-based testing 186  
moderator 83, 84, 86, 102  
module 40, 42  
module test 42  
monitor 216  
multiple condition test 146, 152

**N**

negative test 210  
nonfunctional test 12, 72  
    test tool 216

**P**

partial condition  
    atomic 152  
patch 196  
path test 146, 156  
paths 98  
performance test 72  
phase 41, 42  
point of control 108  
point of observation 108  
portability 11, 13  
priorization 177  
priorization of tests 21, 177, 178  
process standard 202  
product management 200

product risks 188  
product standard 202  
production environment 59, 67  
program statements  
  dead 148  
programming 40  
project 14, 15, 178  
project management 200  
project manager 56  
project plan 56  
project planning 56  
project risks 187

## Q

qualification 174  
qualification profiles 172  
quality assurance plan 174  
quality characteristic 11, 178  
quality goals 183

## R

random test 144  
recorder 87  
recoverability 12  
regression test 200, 219  
regression test capability 214  
release 190–192, 219, 229, 232, 267  
reliability 11, 12  
requirements 6, 9, 21, 22, 40, 58, 60, 70, 176, 177, 207  
requirements based test 70  
requirements definition 40  
requirements management 206  
resources 189  
response time 216  
responsibility 86  
retest 74  
reuse 186  
review 17, 80, 99, 102  
  informal 92  
  introduction 83  
  management 86  
  planning 82

  preparation 83  
  success factors 94  
  technical 91  
review meeting 83  
review team 81–86  
review type 82  
reviewable state 82  
reviewer 87  
rework 85  
risk 15, 16, 188  
risk management 188  
risk-based testing 186  
robustness test 212  
roles 22, 83, 84

## S

safety 16  
safety critical system 16  
script 208, 210  
selection 21, 27, 65, 76  
selection criteria 93  
selection process 220  
side effects of changes 68  
simulator 212  
smoke test 26, 144  
software development models 17  
software failure 12  
software maintenance 65  
software quality 11  
software quality and testability 11  
software test  
  foundations 33  
  general principles 5  
  terms and motivation 6  
source code 23, 209  
specification 16, 22, 48  
spiral model 17  
stability 13  
standard software 62  
standards 97, 202  
state machine 128  
state transition model 128, 130  
state-based test 186

- statement 146
  - statement coverage 146, 148
  - statement test 146
  - static analysis 13, 95, 97
  - static test 53, 79
    - tool 210
  - status follow-up 85
  - stress test 72
  - structural test 74
  - structure-based test 74
  - structured walkthrough 79, 89
  - stub 105
  - syntax test 144
  - system architecture 40, 56
  - system design
    - functional 40
    - technical 40
  - system requirements 206
  - system test 10, 15, 41, 58, 64, 66, 171
  - system test practice 60
  - system, safety critical 15
- T**
- technical review 92
  - technical system design 40
  - test 8
    - alpha 64
    - beta 64
    - blocked 190
    - business process-based 11
    - change related 74
    - checklist oriented 187
    - code-based 145
    - complete (exhaustive) 13, 20
    - component 15, 41 ff, 52, 54, 55, 77, 171
    - control flow-based 13, 14, 16
    - data flow-based 16
    - developer 31
    - dynamic 105
    - evaluation 28
    - field 64
    - functional 70
    - load 10, 72, 216
    - module 42
    - negative 210
    - nonfunctional 12, 72
    - of conditions 8, 25
    - performance 72, 216
    - priorization of 21, 177, 178
    - qualification regression 172, 190, 208, 219
    - random 144
    - regression testable 74, 219
    - requirements-based 70, 71, 207
    - risk-based 188
    - robustness 212
    - smoke 144
    - standards 202
    - state-based 135, 186
    - static 53
    - stress 72
    - structural 74
    - structure-based 74
    - syntax 144
    - system 10, 41, 60 ff, 171, 212, 217, 241, 262, 272
    - task 172
    - unit 42
    - unnecessary 16
    - user acceptance 10, 31, 62, 63
    - volume 72
  - test activities
    - test closure 30
  - test administrator 173
  - test analysis 19, 22, 42
  - test approach 183, 184
    - analytic vs. heuristic 186
    - expert oriented 187
    - preventative vs. reactive 185
  - test automation 75
  - test automator 173, 214
  - test basis 22, 23, 71, 209
  - test case
    - concrete 23, 25, 139, 263
    - fault revealing 261
    - logical 23, 25, 139, 263
    - unnecessary 120, 279
  - test case based metrics 190

- test case design
  - experience based 109
  - intuitive 109
- test case explosion 16
- test case specification 23, 25
- test closure 30
- test control 20, 172
- test cost
  - estimation 184
- test coverage 21, 27, 29, 43, 133, 158, 165, 179, 182, 215
- test cycle
  - control 20, 171, 192
  - planning 189
- test data 9
- test data generator 209
- test design 22
- test designer 173
- test documentation 218
- test driver 44, 53–59, 106, 212, 218, 222
- test end 21, 177
- test environment 26, 43, 53, 59, 60, 172, 173, 175, 194, 195, 212, 237, 272–274
- test evaluation 200
- test execution
  - tools 212
- test frame / test harness 25, 206, 212
- test generator 210, 211
- test goal 9, 31, 34, 50, 60, 106, 107, 152
- test implementation 18, 19, 23, 25
- test infrastructure 21, 25, 182
- test intensity 120
- test lab 170
- test leader 172
- test level 3, 10, 18, 39, 41, 233, 234, 237, 274
- test level plan 237
- test log 56, 170, 193, 240, 274
- test logging 171
- test management 169
- test management tool 206
- test manager 172
- test method 25
- test metrics 190
- test object 3, 9, 10, 43, 52, 275
- test object-based metrics 190
- test of compatibility 73
- test of contract acceptance 62
- test of data conversion 73, 217
- test of decisions 148, 149
- test of different configurations 73
- test of documentation 73
- test of maintainability 73
- test of new product versions 65
- test of reliability 73
- test of robustness 73
- test of security 73
- test of usability 73
- test of user acceptance 63, 279
- test oracle 23, 37
- test organization 169
- test person 10
- test plan
  - after IEEE 829 176, 227
- test planning 19, 20, 22, 23, 174, 192
- test planning work 175
- test prioritization 177
- test process 9
  - the fundamental 17
- test progress 172, 190, 191, 198
- test result 190
- test robot 208, 212, 218, 222
- test scenario 9
- test script 208, 214
  - regression able 214
- test sequence 26
- test specification 22
  - tool 209
- test start criteria 179
- test status report 190
- test summary report (or test report) 30, 201
- test team 32, 169
- test technique 5
- test tool 205
  - introduction 218
  - selection 218
  - types 205
- test type 10

test work  
    management 189  
test-driven development 49, 109  
test-first programming 49  
testability 182  
tester 7, 28, 35, 173, 182  
testing  
    change related 67  
    command-driven 214  
    cost aspects 180  
    data-driven 214  
    economic aspects in the life cycle 39  
    keyword-driven / action-word-driven 214  
    maintenance related 65–67  
    model-based 210  
    new releases 68  
    risk 187  
    with incremental development 68  
testing psychology 31  
tools  
    CASE 206  
    CAST 206  
    economy for dynamic tests 211  
    for management and control of tests 206  
    for nonfunctional test 216  
    for static test 210  
    for test execution 208, 211  
    for test specification 209, 218  
tool chain 208  
tool introduction economy 219, 221  
tool selection 220  
tool support 160  
tool type 69

top-down integration 56  
transaction 216

## U

UML – unified modeling language 141  
understandability 12  
unit test 42  
unnecessary test 16  
unnecessary test case 120  
update 68  
usability 11, 12, 73  
use case-based test 71, 141  
use-case-diagram 142  
user 40  
user interface 212  
user test 10

## V

V-model 18  
    generic 76  
validation 41  
verification 41  
version 65, 200, 201  
version management 201  
volume test 72

## W

walkthrough 89, 102  
waterfall model 18  
white box technique 145

